Proceedings of

SAT RACE 2019

Solver and Benchmark Descriptions

Marijn J. H. Heule, Matti Järvisalo, and Martin Suda *(editors)*

University of Helsinki
Department of Computer Science
Series of Publications B
Report B-2019-1

Helsinki 2019

PREFACE

The area of Boolean satisfiability (SAT) solving has seen tremendous progress over the last years. Many problems (e.g., in hardware and software verification) that seemed to be completely out of reach a decade ago can now be handled routinely. Besides new algorithms and better heuristics, refined implementation techniques turned out to be vital for this success. To keep up the driving force in improving SAT solvers, SAT solver competitions provide opportunities for solver developers to present their work to a broader audience and to objectively compare the performance of their own solvers with that of other state-of-the-art solvers.

SAT Race 2019 (SR 2019; `http://sat-race-2019.ciirc.cvut.cz`), a competitive event for SAT solvers, was organized as a satellite event of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT 2019), Lisbon, Portugal. SR 2019 stands in the tradition of the previously organized main competitive events for SAT solvers: the SAT Competitions held 2002-2005, biannually during 2007-2013, and 2014, 2016-2018; the SAT Races held in 2006, 2008, 2010, and 2015; and SAT Challenge 2012.

Compared to the more diverse SAT Competitions, SAT Race 2019 was a lean competitive event that evaluates the state-of-the-art SAT solvers. In contrast to SAT Competitions, SAT Races consist of only one track that is comparable to the Main Track of the SAT Competitions.
There were two ways of contributing to SR 2019: by submitting one or more solvers for evaluation in the competition and by submitting interesting benchmark instances on which the submitted solvers could be evaluated on in the competition. Following the tradition put forth by SAT Challenge 2012, the rules of SR 2019 invited all contributors to submit a short, 1-2 page long description as part of their contribution. This book contains these non-peer-reviewed descriptions in a single volume, providing a way of consistently citing the individual descriptions.

Successfully running SR 2019 would not have been possible without active support from the community at large. We would like to thank the StarExec initiative (`http://www.starexec.org`) for the computing resources needed to run SR 2019. Many thanks go to Aaron Stump for his invaluable help in setting up StarExec to accommodate for the competition's needs. Furthermore, we would like to emphasize that a competition does not exist without participants: we thank all those who contributed to SR 2019 by submitting either solvers or benchmarks and the related description.

Marijn J. H. Heule, Matti Järvisalo, & Martin Suda
SAT Race 2019 Organizers

Contents

Benchmark Descriptions

SOLVER DESCRIPTIONS

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

CaDiCaL at the SAT Race 2019

Armin Biere
Institute for Formal Models and Verification
Johannes Kepler University Linz

Our SAT solver CaDiCaL provides a clean, documented, easy to understand and modify state-of-the-art solver, based on CDCL [1] with inprocessing [2]. Earlier versions participated in the SAT competition 2017 and 2018. Here we only describe differences to these versions [3], [4]. Even though CaDiCaL performed well on unsatisfiable instances in the SAT Competition 2018, the performance on satisfiable instances was behind the top solvers in that competition. Thus a large part of the changes made and described in this note are motivated by trying to improve CaDiCaL on satisfiable instances without loosing its good performance on unsatisfiable instances.

Separate Decision Queue

The earlier versions of CaDiCaL already partially followed the advice given by Chanseok Oh in [5] to interleave (what we call) *stable* search phases focusing on satisfiable instances with almost no restarts and (again in our terminology) *unstable* search phases with the usual frequent but limited restarts schedule. In our new version we use a reluctant doubling scheme with base conflict interval 1024 for the stable phase.

However, the results of [5] also suggest to use a smoother increase of scores for the stable phase using a separate decision queue. We have integrated this idea. It required to add the usual exponential VSIDS scoring mechanism using a binary heap as in MiniSAT [6]. Thus this new version relies on its previous VMTF queue [7] only for the unstable search phases and on the exponential VSIDS for the stable search phase.

The "default" configuration submitted to the competition alternates stable and unstable phases, while the "unsat" configuration remains in the unstable search phase and the "sat" configuration vice versa only in the "stable" phase.

Local Search

Our local search solver YalSAT [8] solved 48 instances in the main track of the SAT Competition 2018 from which 30 instances were not solved by CaDiCaL and even one not solved by any other solver. It further solved 36 instances faster than any other solver. This shows that it should be beneficial to add a local search component to CaDiCaL. We already had YalSAT hooked up to Lingeling, which was successfully used in Treengeling in parallel solver threads. However controlling the amount of time allocated to YalSAT is difficult. It also requires to copy all clauses.

Therefore we added a simple local search component to CaDiCaL. As YalSAT it is based on ideas developed in ProbSAT [9]. In contrast to YalSAT and ProbSAT, we watch one literal in each clause instead of using counters. The broken (unsatisfiable) clauses are kept on a stack and traversed completely during each step (flipping a literal).

Local search is called from the *rephase* procedure [3], [4] which is scheduled in regular intervals. It can also be executed as preprocessing step for an arbitrary number of rounds, which in essence turns the solver into a local search solver (disabled by default). As initial assignment for local search we use the same assignments that would be selected in the CDCL loop for decision variables (actually the target phases—see next section—are always preferred, even for local search during unstable phases). The best assignment (falsifying the smallest number of clauses) determined during each local search round is exported back to the CDCL loop as saved phases.

Target and Best Phases

Probably the most important new technique is the use of *target phases*, which can be seen as a generalization of phase saving [10]. This well-known technique saves the last value assigned to a variable (its saved phase) and uses it as assignment value if a variable is selected as decision.[1]

In addition to these saved phases our new approach now also maintains an array of target and another array of best phases. The idea is to maximize the size of the trail without conflicts. Thus during backtracking the prefix of the trail is determined which did not (yet) lead to a conflict previous propagations. The values of the literals on the prefix are then saved as new target phases if this prefix is larger than the previously saved one. In stable search phases these target phases are preferred over saved phases [10] for decisions.

During *rephasing* [3], [4] saved phases are reset as before, except, that beside the new local search rephasing discussed above we have further a new *best* rephasing, which sets saved phases to the values of the largest previously reached trail without conflict and then resets these best phases. By default best rephasing is only performed during stable search phases.

Lucky Phases

Occasionally applications produce trivial formulas in the sense that they can be satisfied by for instance assigning all variables to false. Some of them also made it into the competition and therefore we implemented in Lingeling [11]

Supported by FWF, NFN Grant S11408-N23 (RiSE).

[1] Unfortunately there are now two uses of the word "phase" here, one for stable and unstable search phases, as well as for the values assigned to variables. We hope it is clear from the context which of the two interpretation is meant whenever we use "phase".

a "lucky phase" detector. This has been ported to CaDiCaL and extended to detect horn clause benchmarks, which can be satisfied by assigning in forward or backward order all variables to the same constant (interleaved with propagation). For instance satisfiable multiplier miters [12] comparing correct and buggy multipliers can be satisfied by this new lucky phase procedure instantly if the inputs either appear consecutively at the beginning or at the end of the variable range.

Improvements to Preprocessing

Since (bounded) variable elimination [13] remains the most important pre- and inprocessing technique, we tried to improve its effectiveness even further. First, if variable elimination completed, the bound on the number of allowed zero additional clauses (difference between non-tautological resolvents and clauses with a candidate variable) is increased (exponentially from the default zero to 1,2,4,8,16) and all variables are again considered as candidates for elimination attempts. We further perform variable elimination by substitution [13] if we are able to extract AND or XOR gates. We also added eager backward subsumption and strengthening after each successful variable elimination, in addition to our fast forward subsumption algorithm [3] which is continued to be applied to redundant clauses too. Last we added a resolution limit, to reduce the time spent in variable elimination for large but easy to solve formulas. In the same spirit we limit the number of subsumption checks during forward subsumption.

As in previous versions the solver triggers failed literal *probing* (including hyper binary resolution and equivalent literal substitutions) independently from both *subsumption* (on redundant and irredundant clauses followed by vivification) and variable *elimination* (elimination rounds are interleaved with subsumption and optionally, but disabled by default, with blocked and covered clause elimination). These preprocessors can also be called for multiple rounds initially. Using a conflict limit this allows the solver to be used as a CNF preprocessor (the extension stack needed for solution reconstruction can be extracted as well).

Chronological Backtracking

The winner Maple_LCM_Dist_ChronoBT [14] of the main track in the SAT Competition 2018 implemented a combination of chronological backtracking with CDCL [15]. We have ported this idea to CaDiCaL and as in the original work backtrack chronologically if backjumping would jump over more than 100 levels, but otherwise do not limit its application. We further combine it with the idea of reusing the trail [16]. More details will appear in [17].

Incremental Solving and Model Based Testing

Finally we added a new approach [18] to incremental SAT solving which does not require to freeze variables (as in MiniSAT and Lingeling) in order to be combined with inprocessing. To implement such a combination correctly requires sophisticated API testing and accordingly we implemented a tightly integrated model based tester called Mobical following the principles reported in [19].

License

References

[1] J. P. M. Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185, pp. 131–153.

[2] M. Järvisalo, M. Heule, and A. Biere, "Inprocessing rules," in *IJCAR*, ser. Lecture Notes in Computer Science, vol. 7364. Springer, 2012, pp. 355–370.

[3] A. Biere, "CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017," in *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, ser. Dept. of Comp. Science Series of Publications B, vol. B-2017-1. Univ. of Helsinki, 2017, pp. 14–15.

[4] ——, "CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2018," in *Proc. of SAT Competition 2018 – Solver and Benchmark Descriptions*, ser. Dept. of Comp. Science Series of Publications B, vol. B-2018-1. Univ. of Helsinki, 2018, pp. 13–14.

[5] C. Oh, "Between SAT and UNSAT: the fundamental difference in CDCL SAT," in *SAT*, ser. Lecture Notes in Computer Science, vol. 9340. Springer, 2015, pp. 307–323.

[6] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, ser. Lecture Notes in Computer Science, vol. 2919. Springer, 2003, pp. 502–518.

[7] A. Biere and A. Fröhlich, "Evaluating CDCL variable scoring schemes," in *SAT*, ser. Lecture Notes in Computer Science, vol. 9340. Springer, 2015, pp. 405–422.

[8] A. Biere, "Yet another local search solver and Lingeling and friends entering the SAT Competition 2014," in *Proc. SAT Competition 2014, Solver and Benchmark Descriptions*, ser. Dept. of Comp. Science Series of Publications B, vol. B-2014-2. Univ. of Helsinki, 2014, pp. 39–40.

[9] A. Balint and U. Schöning, "Choosing probability distributions for stochastic local search and the role of make versus break," in *SAT*, ser. Lecture Notes in Comp. Science, vol. 7317. Springer, 2012, pp. 16–29.

[10] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *SAT*, ser. Lecture Notes in Computer Science, vol. 4501. Springer, 2007, pp. 294–299.

[11] " Lingeling and Friends Entering the SAT Race 2015," FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, Tech. Rep., 2015.

[12] A. Biere, "Collection of Combinational Arithmetic Miters Submitted to the SAT Competition 2016," in *Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions*, ser. Dept. of Computer Science Series of Publications B, vol. B-2016-1. Univ. of Helsinki, 2016, pp. 65–66.

[13] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *SAT*, ser. Lecture Notes in Computer Science, vol. 3569. Springer, 2005, pp. 61–75.

[14] A. Nadel and V. Ryvchin, "Maple_LCM_Dist_ChronoBT: Featuring chronological backtracking," in *Proc. of SAT Competition 2018 – Solver and Benchmark Descriptions*, ser. Dept.Dept. Computer Science Series of Publications B, vol. B-2018-1. Univ. of Helsinki, 2018, p. 29.

[15] ——, "Chronological backtracking," in *SAT*, ser. Lecture Notes in Computer Science, vol. 10929. Springer, 2018, pp. 111–121.

[16] P. van der Tak, A. Ramos, and M. Heule, "Reusing the assignment trail in CDCL solvers," *JSAT*, vol. 7, no. 4, pp. 133–138, 2011.

[17] S. Möhle and A. Biere, "Backing backtracking," 2019, submitted.

[18] K. Fazekas, A. Biere, and C. Scholl, "Incremental inprocessing in SAT solving," 2019, submitted.

[19] C. Artho, A. Biere, and M. Seidl, "Model-based testing for verification back-ends," in *TAP*, ser. Lecture Notes in Computer Science, vol. 7942. Springer, 2013, pp. 39–55.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

Candy for SAT Race 2019

Markus Iser* and Felix Kutzner†
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
*markus.iser@kit.edu, †felix@kutzner.io

***Abstract*—We use Candy as a platform to systematically analyse the properties of competing strategies in a portfolio.**

I. Introduction

Candy [1] is a fork of **Glucose 3** [5], [6]. Candy provides a flexible and efficient architecture to experiment with many different strategies. The solver does this by orchestrating a set of loosely coupled systems, which mainly provide an interface for implementations of competing strategies. For example, Candy provides a variety of strategies in the branching-system.

II. Implementation

Among others, the branching system can resort to implementations of gate-analysis and random-simulation based *implicit learning* (RSIL) which uses the algorithms which we presented in [7] and [8].

Candy now also has a parallel mode with selected combinations and configurations of strategies and efficient clause sharing.

Candy implements the IPASIR interface [3] and an interface to the generic massively parallel SAT solver HordeSAT [2]. The sonification interface makes solver runs even audible [4].

III. Candy in SAT Race 2019

We submitted Candy in its default setting which is a configuration of strategies that is roughly similar to the one used in Glucose 3. This is the public evaluation of our baseline performance for reference.

References

[1] Candy GIT. https://github.com/Udopia/candy-kingdom
[2] HordeSAT GIT. https://github.com/biotomas/hordesat
[3] Ipasir GIT. https://github.com/biotomas/ipasir
[4] MiniSAT Sonification. https://www.youtube.com/watch?v=iupgZGlzMCQ
[5] Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Proceedings of the 21st International Jont Conference on Artifical Intelligence. pp. 399–404. IJCAI'09, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2009)
[6] Eén, N., Sörensson, N.: An extensible sat-solver. In: Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers. pp. 502–518 (2003)
[7] Iser, M., Kutzner, F., Sinz, C.: Using gate recognition and random simulation for under-approximation and optimized branching in SAT solvers. In: 29th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2017, Boston, MA, USA, November 6-8, 2017. pp. 1029–1036 (2017)
[8] Iser, M., Manthey, N., Sinz, C.: Recognition of nested gates in CNF formulas. In: Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings. pp. 255–271 (2015)

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

CCAnrSim in SAT Race 2019

Md Shibbir Hossen
Dept. of Computer Science & Engineering
Jagannath University
Dhaka, Bangladesh
shibbir.hossen@live.com

Md Masbaul Alam Polash
Dept. of Computer Science & Engineering
Jagannath University
Dhaka, Bangladesh
mdmasbaul@gmail.com

***Abstract*—This document is provided as a brief description of the SAT solver named "CCAnrSim" which is a stochastic local search (SLS) solver.**

***Index Terms*—SAT, Local Search, Similarity Checking Meta-heuristics**

I. Introduction

A very common problem of local search is it regenerates the same solution after a certain time. Therefore, search may falls in a stagnation stage. Here, we have introduced a new method named Similarity Checking (SC) to escape from stagnation stage. SC was a successful method in biological science for protein structure prediction and that can be found in [1]. SC metaheuristic is implemented on a comparable solver for SAT named CCAnr [2].

II. Main Technique

CCAnrSim is an SLS based SAT solver and it incorporates common local search techniques like tabu, aspiration and clause weighting. It includes CCA heuristic [3] that is used to pick a compromising variable. To overcome the regeneration of same solutions after a certain time, we have introduced a SC meta-heuristic. The main technique of SC is to check similarity of the current solution with some previously saved solutions. If the similarity is high then this solution is discarded and applied a focused random walk.The $pickVar()$ function in [2] returns the variable to be flipped next.

Algorithm 1 describes the pseudo-code of similarity checking. At line 1, we store some previously visited solutions within a custom interval in $elite\_sol[]$. At line 2, $elite\_iter$ indicates the last solution index. So whenever a new solution is found, we will check against all previously stored solutions. It is not a good idea to save all previously visited solutions because of memory constraint and that's why we save the solutions after a custom interval. At line 5, we determine the similarity of current solution against all stored solutions and if the similarity exceeds a predefined proximity value, we discard the solution. If it is discarded then it performs a focused random-walk to return a variable. Otherwise, we will store it and return $flipVar$ that was picked from $pickVar$.

III. Main parameters

In our solver, we set similarity percentage of $proximity = 90\%$ and number of elite solution $elite\_sol\_length = 20$. We employ the Algorithm 1 after each 100 iterations.

Algorithm 1 $similarity\_checking(flipVar)$

```
1:  elite_sol[] stores previously visited solutions
2:  elite_itr is the last solution index at elite_sol[]
3:  i ← 1
4:  while ++i ≤ elite_itr do
5:      if similarity(cur_sol, elite_sol[i]) ≥ proximity then
6:          discard this solution
7:          break
8:  if i = elite_itr then
9:      elite_itr++
10:     elite_sol[elite_itr] ← current_sol
11:     return flipVar
12: else
13:     flipVar ← pick variable by performing focused random-walk mode
14:     return flipVar
```

IV. IMPLEMENTATION DETAILS

Here, we have used C++ language to build the solver. It is based on the code of CCAnr [2].

V. Running Technique

This solver is submitted to SAT Race 2019 for Application and Hard-combinatorial tracks. It is compiled by g++ compiler. Running command is :
$./CCAnrSim\ -\ inst\ <\ instance\ name\ >$

Acknowledgment

We are thankful to Dr. Shaowei Cai and Prof. Kaile Su and for sharing their strategies and source codes. We also thankful to Ministry of ICT of Bangladesh to provide us funding for developing this solver.

References

[1] S. Shatabda, M. Newton, D. N. Pham, and A. Sattar, "Memory-based local search for simplified protein structure prediction," in *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine*. ACM, 2012, pp. 345–352.

[2] S. Cai, C. Luo, and K. Su, "Ccanr: A configuration checking based local search solver for non-random satisfiability," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2015, pp. 1–8.

[3] S. Cai and K. Su, "Configuration checking with aspiration in local search for sat." in *AAAI*, 2012.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

CryptoMiniSat 5.6 with WalkSAT at the SAT Race 2019

Mate Soos (National University of Singapore)
Bart Selman (Cornell University), Henry Kautz (University of Rochester)

I. Introduction

This paper presents the conflict-driven clause-learning (CLDL) SAT solver CryptoMiniSat v5.6 (*CMS*) augmented with the Stochastic Local Search (SLS) [10] solver WalkSAT v56 as submitted to SAT Race 2019.

CryptoMiniSat aims to be a modern, open source SAT solver using inprocessing techniques, optimized data structures and finely-tuned timeouts to have good control over both memory and time usage of inprocessing steps. It also supports, when compiled as such, to recover XOR constraints and perform Gauss-Jordan elimination on them at every decision level. For the competition, this option was disabled. CryptoMiniSat is authored by Mate Soos.

WalkSAT [6] is a standard system to solve satisfiability problems using Stochastic Local Search. The version inside CryptoMiniSat is functionally equivalent to the "rnovelity" heuristic of WalkSAT v56 using an adaptive noise heuristic [4]. It behaves exactly as WalkSAT with the minor modification of performing early-abort in case the "low-bad" statistic (i.e. the quality indicator of the current best solution) indicates the solution is far. In these cases, we early abort, let the CDCL solver work longer to simplify the problem, and come back to WalkSAT later. The only major modification to WalkSAT has been to allow it to import variables and clauses directly from the main solver taking into account assumptions given by the user.

A. Composing the Two Solvers

The two solvers are composed together in a way that does *not* resemble portfolio solvers. The system runs the CDCL solver CryptoMiniSat, along with its periodic inprocessing, by default. However, at every N inprocessing step, CryptoMiniSat's irredundant clauses are pushed into the SLS solver (in case the predicted memory use is not too high). The SLS solver is then allowed to run for a predefined number of steps. In case the SLS solver finds a solution, this is given back to the CDCL solver, which then performs all the necessary extension to the solution (e.g. for Bounded Variable Elimination, BVE [2]) and then outputs the solution.

Note that the inclusion of the SLS solver is full in the sense that assumptions-based solving, library-based solver use, and all other uses of the SAT solver is fully supported with SLS solving enabled. Hence, this is not some form of portfolio where a simple shell script determines which solver to run and then runs that solver. Instead, the SLS solver is a full member of the CDCL solver, much like any other inprocessing system, and works in tandem with it. For example, in case an inprocessing step has reduced the number of variables through BVE or increased it through BVA [8], the SLS solver will then try to solve the problem thus modified. In case the SLS solver finds a solution, the main solver will then correctly manipulate is to fit the needs of the "outside world", i.e. the caller.

As the two solvers are well-coupled, the combination of the two solvers can solve problems that neither system can solve on its own. Hence, *the system is more than just a union of its parts* which is not the case for traditional portfolio solvers.

II. Major Improvements

A. Via Negativa

The system has been subjected to a thorough investigation whether all the different systems that have been implemented into it actually make the solver faster. In this spirit, failed literal probing [7], stamping [3], burst searching (random variable picking), and blocked clause elimination [5] have all been disabled.

B. Chronological Backtracking

Chronological backtracking [9] has been implemented into a branch of the solver. However, chronological backtracking (CBT) is a double-edged sword. Firstly, it slows down the solver's normal functionality as it adds a number of expensive checks to both the propagation and the backtracking code. Secondly, it changes the trail of the solver in ways that make it hard to reason about the current state of the solver. Finally, it seems only to help with satisfiable instances which are theoretically less interesting for the author of CryptoMiniSat. These issues make CBT a difficult addition.

Currently, CryptoMiniSat by default does not implement CBT. The SAT Race has two versions submitted, clearly marked, one with, an one without CBT.

C. Cluster Tuning

The author has been generously given time on the ASPIRE-1 cluster of the National Supercomputing Center Singapore[1]. This allowed experimentation and tuning that would have been impossible otherwise. Without this opportunity, CryptoMiniSat would not stand a chance at the SAT Race.

III. General Notes

A. On-the-fly Gaussian Elimination

On-the-fly Gaussian elimination is again part of CryptoMiniSat. This is explicitly disabled for the competition, but the code is available and well-tested. This allows for special

uses of the solver that other solvers, without on-the-fly Gaussian elimination, are not capable of.

B. Robustness

CMS aims to be usable in both industry and academia. CMS has over 150 test cases and over 2000 lines of Python just for fuzzing orchestration, and runs without fault under both the ASAN and UBSAN sanitisers of clang. It also compiles and runs under Windows, Linux and MacOS X. This is in contrast many academic winning SAT solvers that produce results that are non-reproducible, cannot be compiled on anything but a few select systems, and/or produce segmentation faults if used as a library. CryptoMiniSat has extensive fuzzing setup for library usage and is very robust under strange/unexpected use cases.

IV. Thanks

This work was supported in part by NUS ODPRT Grant R-252-000-685-133 and AI Singapore Grant R-252- 000-A16-490. The computational work for this article was performed on resources of the National Supercomputing Center, Singapore[1]. The author would also like to thank all the users of CryptoMiniSat who have submitted over 500 issues and many pull requests to the GitHub CMS repository[11].

References

[1] ASTAR, NTU, NUS, SUTD: National Supercomputing Centre (NSCC) Singapore (2018), https://www.nscc.sg/about-nscc/overview/

[2] Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing. pp. 61–75. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)

[3] Heule, M.J.H., Järvisalo, M., Biere, A.: Efficient CNF simplification based on binary implication graphs. In: Sakallah, K.A., Simon, L. (eds.) Theory and Applications of Satisfiability Testing - SAT 2011. pp. 201–215. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

[4] Hoos, H.H.: An adaptive noise mechanism for WalkSAT. In: Dechter, R., Kearns, M.J., Sutton, R.S. (eds.) Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada. pp. 655–660. AAAI Press / The MIT Press (2002), http://www.aaai.org/Library/AAAI/2002/aaai02-098.php

[5] Järvisalo, M., Biere, A., Heule, M.: Blocked Clause Elimination. In: Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 6015, pp. 129–144. Springer International Publishing (2010)

[6] Kautz, H.A., Selman, B.: Pushing the envelope: Planning, propositional logic and stochastic search. In: Clancey, W.J., Weld, D.S. (eds.) Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 2. pp. 1194–1201. AAAI Press / The MIT Press (1996), http://www.aaai.org/Library/AAAI/1996/aaai96-177.php

[7] Lynce, I., Silva, J.P.M.: Probing-based preprocessing techniques for propositional satisfiability. In: 15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2003), 3-5 November 2003, Sacramento, California, USA. p. 105. IEEE Computer Society (2003), https://doi.org/10.1109/TAI.2003.1250177

[8] Manthey, N., Heule, M.J.H., Biere, A.: Automated reencoding of boolean formulas. In: Biere, A., Nahir, A., Vos, T. (eds.) Hardware and Software: Verification and Testing. pp. 102–117. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

[9] Nadel, A., Ryvchin, V.: Chronological backtracking. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) Theory and Applications of Satisfiability Testing – SAT 2018. pp. 111–121. Springer International Publishing, Cham (2018)

[10] Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. In: DIMACS Series in Discrete Mathematics and Theoretical Computer Science. pp. 521–532 (1995)

[11] Soos, M.: CryptoMiniSat SAT solver GitHub page (2018), https://github.com/msoos/cryptominisat

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

CryptoMiniSat 5.6 with YalSAT at the SAT Race 2019

Mate Soos (National University of Singapore), Armin Biere (JKU Linz)

I. Introduction

This paper presents the conflict-driven clause-learning (CLDL) SAT solver CryptoMiniSat v5.6 (*CMS*) augmented with the Stochastic Local Search (SLS) [11] solver YalSAT 03v as submitted to SAT Race 2019.

CryptoMiniSat aims to be a modern, open source SAT solver using inprocessing techniques, optimized data structures and finely-tuned timeouts to have good control over both memory and time usage of inprocessing steps. It also supports, when compiled as such, to recover XOR constraints and perform Gauss-Jordan elimination on them at every decision level. For the competition, this option was disabled. CryptoMiniSat is authored by Mate Soos.

Yet Another Local Search SAT Solver (YalSAT) implements several variants of ProbSAT's [4] algorithm and recent extensions [3]. These variants are selected randomly at restarts, scheduled by a reluctant doubling scheme (Luby). For further details, see [1]. YalSAT is authored by Armin Biere.

A. Composing the Two Solvers

The two solvers are composed together in a way that does *not* resemble portfolio solvers. The system runs the CDCL solver CryptoMiniSat, along with its periodic inprocessing, by default. However, at every N inprocessing step, CryptoMiniSat's irredundant clauses are pushed into the SLS solver (in case the predicted memory use is not too high). The SLS solver is then allowed to run for a predefined number of steps. In case the SLS solver finds a solution, this is given back to the CDCL solver, which then performs all the necessary extension to the solution (e.g. for Bounded Variable Elimination, BVE [5]) and then outputs the solution.

Note that the inclusion of the SLS solver is full in the sense that assumptions-based solving, library-based solver use, and all other uses of the SAT solver is fully supported with SLS solving enabled. Hence, this is not some form of portfolio where a simple shell script determines which solver to run and then runs that solver. Instead, the SLS solver is a full member of the CDCL solver, much like any other inprocessing system, and works in tandem with it. For example, in case an inprocessing step has reduced the number of variables through BVE or increased it through BVA [9], the SLS solver will then try to solve the problem thus modified. In case the SLS solver finds a solution, the main solver will then correctly manipulate it to fit the needs of the "outside world", i.e. the caller.

As the two solvers are well-coupled, the combination of the two solvers can solve problems that neither system can solve on its own. Hence, *the system is more than just a union of its parts* which is not the case for traditional portfolio solvers.

II. Major Improvements

A. Via Negativa

The system has been subjected to a thorough investigation whether all the different systems that have been implemented into it actually make the solver faster. In this spirit, failed literal probing [8], stamping [6], burst searching (random variable picking), and blocked clause elimination [7] have all been disabled.

B. Chronological Backtracking

Chronological backtracking [10] has been implemented into a branch of the solver. However, chronological backtracking (CBT) is a double-edged sword. Firstly, it slows down the solver's normal functionality as it adds a number of expensive checks to both the propagation and the backtracking code. Secondly, it changes the trail of the solver in ways that make it hard to reason about the current state of the solver. Finally, it seems only to help with satisfiable instances which are theoretically less interesting for the author of CryptoMiniSat. These issues make CBT a difficult addition.

Currently, CryptoMiniSat by default does not implement CBT. The SAT Race has two versions submitted, clearly marked, one with, an one without CBT.

C. Cluster Tuning

The author has been generously given time on the ASPIRE-1 cluster of the National Supercomputing Center Singapore[2]. This allowed experimentation and tuning that would have been impossible otherwise. Without this opportunity, CryptoMiniSat would not stand a chance at the SAT Race.

III. General Notes

A. On-the-fly Gaussian Elimination

On-the-fly Gaussian elimination is again part of CryptoMiniSat. This is explicitly disabled for the competition, but the code is available and well-tested. This allows for special uses of the solver that other solvers, without on-the-fly Gaussian elimination, are not capable of.

B. Robustness

CMS aims to be usable in both industry and academia. CMS has over 150 test cases and over 2000 lines of Python just for fuzzing orchestration, and runs without fault under both the ASAN and UBSAN sanitisers of clang. It also compiles and runs under Windows, Linux and MacOS X.

This is in contrast many academic winning SAT solvers that produce results that are non-reproducible, cannot be compiled on anything but a few select systems, and/or produce segmentation faults if used as a library. CryptoMiniSat has extensive fuzzing setup for library usage and is very robust under strange/unexpected use cases.

IV. Thanks

This work was supported in part by NUS ODPRT Grant R-252-000-685-133 and AI Singapore Grant R-252- 000-A16-490. The computational work for this article was performed on resources of the National Supercomputing Center, Singapore[2]. The author would also like to thank all the users of CryptoMiniSat who have submitted over 500 issues and many pull requests to the GitHub CMS repository[12].

References

[1] Anton, B., Daniel, D., Heule, M.J.H., Jarvisalo, M.: Yet another Local Search Solver and Lingeling and Friends Entering the SAT Competition 2014. In: Proceedings of SAT Competition 2014 (2014)

[2] ASTAR, NTU, NUS, SUTD: National Supercomputing Centre (NSCC) Singapore (2018), https://www.nscc.sg/about-nscc/overview/

[3] Balint, A., Biere, A., Fröhlich, A., Schöning, U.: Improving implementation of SLS solvers for SAT and new heuristics for k-SAT with long clauses. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing – SAT 2014. pp. 302–316. Springer International Publishing, Cham (2014)

[4] Balint, A., Schöning, U.: Choosing probability distributions for stochastic local search and the role of make versus break. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing – SAT 2012. pp. 16–29. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

[5] Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing. pp. 61–75. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)

[6] Heule, M.J.H., Järvisalo, M., Biere, A.: Efficient CNF simplification based on binary implication graphs. In: Sakallah, K.A., Simon, L. (eds.) Theory and Applications of Satisfiability Testing - SAT 2011. pp. 201–215. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

[7] Järvisalo, M., Biere, A., Heule, M.: Blocked Clause Elimination. In: Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 6015, pp. 129–144. Springer International Publishing (2010)

[8] Lynce, I., Silva, J.P.M.: Probing-based preprocessing techniques for propositional satisfiability. In: 15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2003), 3-5 November 2003, Sacramento, California, USA. p. 105. IEEE Computer Society (2003), https://doi.org/10.1109/TAI.2003.1250177

[9] Manthey, N., Heule, M.J.H., Biere, A.: Automated reencoding of boolean formulas. In: Biere, A., Nahir, A., Vos, T. (eds.) Hardware and Software: Verification and Testing. pp. 102–117. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

[10] Nadel, A., Ryvchin, V.: Chronological backtracking. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) Theory and Applications of Satisfiability Testing – SAT 2018. pp. 111–121. Springer International Publishing, Cham (2018)

[11] Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. In: DIMACS Series in Discrete Mathematics and Theoretical Computer Science. pp. 521–532 (1995)

[12] Soos, M.: CryptoMiniSat SAT solver GitHub page (2018), https://github.com/msoos/cryptominisat

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

COMiniSatPS Pulsar in SAT Race 2019

Chanseok Oh
Google
New York, NY, USA

***Abstract*—COMiniSatPS is a patched MiniSat generated by applying a series of small diff patches to the last available version (2.2.0) of MiniSat that was released several years ago. The essence of the patches is to include only minimal changes necessary to make MiniSat sufficiently competitive with modern SAT solvers. One important goal of COMiniSatPS is to provide these changes in a highly accessible and digestible form so that the necessary changes can be understood easily to benefit wide audiences, particularly starters and non-experts in practical SAT. As such, the changes are provided as a series of incrementally applicable diff patches, each of which implements one feature at a time. COMiniSatPS has many variations. The variations are official successors to an early prototype code-named SWDiA5BY that saw great successes in the past SAT-related competitive events.**

I. Introduction

It has been shown in many of the past SAT-related competitive events that very simple solvers with tiny but critical changes (e.g, MiniSat [1] hack solvers) can be impressively competitive or even outperform complex state-of-the-art solvers [2]. However, the original MiniSat itself is vastly inferior to modern SAT solvers in terms of actual performance. This is no wonder, as it has been many years since the last 2.2.0 release of MiniSat. To match the performance of modern solvers, MiniSat needs to be modified to add some of highly effective techniques of recent days. Fortunately, small modifications are enough to bring up the performance of any simple solver to the performance level of modern solvers. COMiniSatPS [3]. adopts only simple but truly effective ideas that can make MiniSat sufficiently competitive with recent state-of-the-art solvers. In the same minimalistic spirit of MiniSat, COMiniSatPS prefers simplicity over complexity to reach out to wide audiences. As such, the solver is provided as a series of incremental patches to the original MiniSat. Each small patch adds or enhances one feature at a time and produces a fully functional solver. Each patch often changes solver characteristics fundamentally. This form of source distribution by patches would benefit a wide range of communities, as it is easy to isolate, study, implement, and adopt the ideas behind each incremental change. The goal of COMiniSatPS is to lower the entering bar so that anyone interested can implement and test their new ideas easily on a simple solver guaranteed with exceptional performance.

The patches first transform MiniSat into Glucose [4] and then into SWDiA5BY. Subsequently, the patches implement new techniques described in [5], [2], and [6] to generate the current form of COMiniSatPS.

COMiniSatPS is a base solver of the MapleCOMSPS solver series [7], [8], [9], [10] that participated in SAT Competition 2016, 2017, 2018, and SAT Race 2019.

II. COMiniSatPS Pulsar

This year's solver is identical to the last year's solver.

III. Availability and License

Source is available for download for all the versions described in this paper. Note that the license of the M4RI library (used to implement the Gaussian elimination) is GPLv2+.

Acknowledgment

We thank specifically the authors of Glucose, GlueMiniSat, Lingeling, CryptoMiniSat, and MiniSat.

References

[1] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, 2003.
[2] C. Oh, "Improving SAT solvers by exploiting empirical characteristics of CDCL," Ph.D. dissertation, New York University, 2016.
[3] ——, "Patching MiniSat to deliver performance of modern SAT solvers," in *SAT-RACE*, 2015.
[4] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *IJCAI*, 2009.
[5] C. Oh, "Between SAT and UNSAT: The fundamental difference in CDCL SAT," in *SAT*, 2015.
[6] ——, "COMiniSatPS the Chandrasekhar Limit and GHackCOMSPS," in *SAT Competition*, 2016.
[7] J. H. Liang, C. Oh, V. Ganesh, K. Czarnecki, and P. Poupart, "MapleCOMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB," in *SAT Competition*, 2016.
[8] ——, "MapleCOMSPS_LRB_VSIDS and MapleCOMSPS_CHB_VSIDS," in *SAT Competition*, 2017.
[9] J. H. Liang, C. Oh, K. Czarnecki, P. Poupart, and V. Ganesh, "MapleCOMSPS_LRB_VSIDS and MapleCOMSPS_CHB_VSIDS in the 2018 Competition," in *SAT Competition*, 2018.
[10] ——, "MapleCOMSPS_LRB_VSIDS and MapleCOMSPS_CHB_VSIDS in the 2019 Competition," in *SAT Race*, 2019.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

Four CDCL SAT Solvers based on Exploration and Glue Variable Bumping

Md Solimul Chowdhury
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
mdsolimu@ualberta.ca

Martin Müller
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
mmueller@ualberta.ca

Jia-Huai You
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
jyou@ualberta.ca

***Abstract*—We describe four CDCL SAT solvers: MLD-ChronoBT_GCBump, expMaple_CM, expMaple_CM_GCBump, expMaple_CM_GCBumpOnlyLRB, which are entering the SAT Race-2019. These solvers are based on two new ideas: 1) Guidance of VSIDS via random exploration during conflict depression phases and 2) Activity score bumping of Glue variables.**

I. Guidance of VSIDS via random exploration during conflict depression

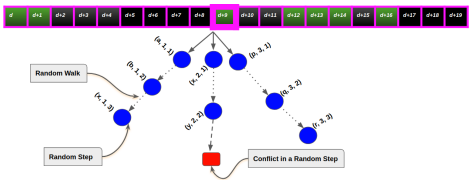
This approach is based on our observation that CDCL SAT solving entails clear non-random patterns of bursts of conflicts followed by longer phases of *conflict depression (CD)*. During a CD phase a CDCL SAT solver is unable to generate conflicts for a consecutive number of decisions. To correct the course of such a search, we propose to use exploration to combat conflict depression. We therefore design a new SAT solver, called *expSAT*, which uses random walks in the context of CDCL SAT solving. In a conflict depression phase, random walks help identify more promising variables for branching. As a contrast, while exploration explores *future* search states, VSIDS relies on conflicts generated from the *past* search states.

A. expSAT algorithm

Given a CNF SAT formula $\mathcal{F}$, let $vars(\mathcal{F})$, $uVars(\mathcal{F})$ and $assign(\mathcal{F})$ denote the set of variables in $\mathcal{F}$, the set of currently unassigned variables in $\mathcal{F}$ and the current partial assignment, respectively. In addition to $\mathcal{F}$, *expSAT* also accepts four *exploration parameters* nW, lW, p_{exp} and ω, where $1 \leq nW, lW \leq uVars(\mathcal{F})$, $0 < p_{exp}, \omega \leq 1$. These parameters control the exploration aspects of *expSAT*. The details of these parameters are given below.

Given a CDCL SAT solver, *expSAT* modifies it as follows: (I) Before each branching decision, if a **substantially large CD phase** is detected then with probability p_{exp}, *expSAT* performs an *exploration episode*, consisting of a fixed number *nW* of random walks. Each walk consists of a limited number of *random steps*. Each such step consists of (a) the uniform random selection of a currently unassigned *step variable* and assigning a boolean value to it using a standard CDCL *polarity* heuristic, and (b) a followed by Unit Propagation (UP). A walk terminates either when a conflict occurs during UP, or after a fixed number *lW* of random steps have been taken. Figure 1 illustrates an exploration episode amid a CD phase. (II) In an exploration episode of nW walks of maximum length lW, the *exploration score expScore* of a decision variable v is the average of the *walk scores* $ws(v)$ of all those random walks within the same episode in which v was one of the randomly chosen decision variables. $ws(v)$ is computed as follows: (a) $ws(v) = 0$ if the walk ended without a conflict. (b) Otherwise, $ws(v) = \frac{\omega^d}{lbd(c)}$, with decay factor $0 < \omega \leq 1$, $lbd(c)$ the LBD score of the clause c learned for the current conflict, and $d \geq 0$ the *decision distance* between variable v and the conflict which ended the current walk: If v was assigned at some step j during the current walk, and the conflict occurred after step $j' \geq j$, then $d = j' - j$. We assign credit to all the step variables in a walk that ends with a conflict and give higher credit to variables closer to the conflict. (III) The novel branching heuristic `expVSIDS` adds VSIDS score and *expScore* of the unassigned variables. At the current state of the search, the variable bumping factor of VSIDS is g^z, where $g > 1$ and $z \geq 1$ is the count of conflicts in the search so far. To achieve a comparable scale for $expScore$ and VSIDS score, we scale up the *expScore* by g^z before adding these scores. A variable v^* with maximum combined score is selected for branching. (IV) All other components remain the same as in the underlying CDCL SAT solver.

Fig. 1: The 20 adjacent cells denote 20 consecutive decisions starting from the d^{th} decision, with $d > 0$, where a green cell denotes a decision with conflicts and a black cell denotes a decision without conflicts. Say that amid a CD phase, just before taking the $(d + 9)^{th}$ decision, *expSAT* performs an exploration episode via 3 random walks each limited to 3 steps . The second walk ends after 2 steps, due to a conflict. A triplet (v, i, j) represents that the variable v is randomly chosen at the j^{th} step of the i^{th} walk.



The Parameter Adaptation Algorithm: From the empirical perspective, a parameter setting that is effective for one instance may not be effective for another. To address this issue, here we use an adaptive algorithm *paramAdapt* to dynamically control when to trigger exploration episodes, and how much exploration to perform in an exploration episode. The three exploration parameters p_{exp}, nW, and lW are adapted between CDCL restarts based on the search behavior. The details of *paramAdapt* is given below:

The search in *expSAT* starts with a initial value P^{init} of P. $paramAdapt$ keeps track of the following statistics about all exploration steps within a period: the number of random steps $rSteps$, the number of conflicts c, the number of glue-clauses gc, the mean LBD value, lbd, of the learned clauses.

With fixed weights $w_1 > w_2 > w_3$, an *exploration performance metric* (EPM) is defined as

$$\frac{w_1 \times gc + w_2 \times c}{rSteps} + w_3 * \frac{1}{lbd}$$

This performance metric rewards finding glue clauses (most important), finding any conflict (very important), and learning clauses with low LBD score (important).

At each restart, the algorithm computes a new EPM σ^{new} and compares (the comparison starts after the second restart) it with the prior one σ^{old}, and update the parameter setting P^{old} just used to get a new setting P^{new}.

- If $\sigma^{new} < \sigma^{old}$, the performance of exploration is worse than before. First, P^{new} is set to the old P^{old}, then we perform an *increment*: Randomly select a parameter $p \in P$ and increase its value by a predefined stepsize.
- If $\sigma^{new} = \sigma^{old}$, we only perform the *increment*, no reset.
- If $\sigma^{new} > \sigma^{old}$, then exploration is working better than before. We do not change P^{old} in this case.

The values of a parameter are bounded by a range. Whenever a value leaves its range, it is reset to its initial value.

II. Glue Variable Bumping

Let a CDCL SAT solver M is running a given SAT instance $\mathcal{F}$ and the current state of the search is S. We call the variables that appeared in at least one glue clause up to the current state S *Glue Variables*. We design a structure-aware variable score bumping method named *Glue Bumping* (GB), based on the notion of *glue centrality (gc)* of glue variables. Given a glue variable v_g, glue centrality of v_g dynamically measures the fraction of the glue clauses in which v_g appears, until the current state of the search. Mathematically, the glue centrality of v_g, $gc(v_g)$ is defined as follows:

$$gc(v_g) \leftarrow \frac{gl(v_g)}{ng}$$

, where ng is the total number of glue clauses generated by the search so far. $gl(v_g)$ is the glue level of v_g, a count of glue clauses in which v_g appears, with $gl(v_g) \leq ng$.

A. The GB Method

The GB method modifies a CDCL SAT solver M by adding two procedures to it, named *Increase Glue Level* and *Bump Glue Variable*, which are called at different states of the search. We denote by M^{gb} the GB extension of the solver M.

Increase Glue Level: Whenever M^{gb} learns a new glue clause g, before making an assignment with the first UIP variable that appears in g, it invokes this procedure. For each variable v_g in g, its glue level, $gl(v_g)$ is increased by 1.

Bump Glue Variable: This procedure bumps a glue variable v_g, which has just been unassigned by backtracking. First a bumping *factor (bf)* is computed as follows:

$$bf \leftarrow activity(v_g) * gc(v_g)$$

, where $activity(v_g)$ is the current activity score of the variable v_g and $gc(v_g)$ is the glue centrality of v_g.

Finally, the activity score of v_g, $activity(v_g)$ is bumped as follows:

$$activity(v_g) \leftarrow activity(v_g) + bf$$

III. Solvers Description

We have submitted four CDCL SAT solvers to SAT Race-2019, which are based on four combinations of the two approaches described in the previous two sections. Our solvers are implemented on top of the solvers *MapleLCMDistChronoBT* and *Maple_CM*. In the following, we describe our solvers:

a) ***MLDChronoBT_GCBump****:* This solver extends MapleLCMDistChronoBT by implementing the the GB method. In MLDChronoBT_GCBump, Glue Variables are bumped for all its three heuristics, namely- Dist, LRB and VSIDS.

b) ***expMaple_CM****:* The corresponding baseline system Maple_CM has three switches between VSIDS and LRB (i) it runs VSIDS for the first 10,000 conflicts, (ii) then switches to LRB, which runs until 2,500 seconds, and (iii) then switches to VSIDS for the rest of the execution of the solver. In expMaple_CM, we replace VSIDS with *expVSIDS* for phase (iii) and have kept everything else the same as in Maple_CM.

c) ***expMaple_CM_GCBump****:* expMaple_CM_GCBump is an GB extension of expMaple_CM. This solver performs Glue variable bumping for both LRB and `expVSIDS`.

d) ***expMaple_CM _GCBumpOnlyLRB****:* This system is a variant of expMaple_CM_GCBump, where GB method only increases the activity score of LRB heuristic In this solver, the glue variable bumping is turned off after 2,500 seconds.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

Glucose in the SAT Race 2019

Gilles Audemard
Univ. Lille-Nord de France
CRIL/CNRS UMR8188
audemard@cril.fr

Laurent Simon
Univ. Bordeaux, CNRS, Bordeaux INP,
LaBRI, UMR 5800
F-33400, Talence, France
lsimon@labri.fr

***Abstract*—Glucose is a CDCL solver developed on top of Minisat almost ten years ago, with a special focus on removing useless clauses as soon as possible, and an original restart scheme based on the quality of recent learnt clauses. We describe in this short description the small novelties introduced for the SAT 2018 competition, that remains in the 2019 edition. We also give some experimental evidences that motivated our choice to maintain Glucose as is.**

I. Introduction

Glucose is a CDCL (Conflict Driven Clause Learning) solver introduced in 2009 that tries to center all the components of the SAT solver around a measure of learnt clause quality, called LBD, for Literal Block Distance. This measure allows to delete a lot of learnt clauses from the beginning of the computation. From a practical point of view, it seems that this feature allows Glucose to produce more shorter proofs, which probably explains why Glucose and Syrup won a number of competitions in the last 9 years. A recent survey paper summarises most of the improvements we added to the original Glucose [1]. Of course, the current short description does not mean to be exhaustive and the interested reader should refer to the previous paper.

In a few words, however, Glucose enters SAT competitions/races [2], [3] every year since its creation. Glucose is based on the internal architecture of Minisat [4] (especially for the VSIDS implementation, the 2-Watched scheme and the memory management of clauses (garbage collection, ...)). It is based on the notion of Literal Block Distance, as aforementioned, a measure that is able to estimate the quality of learnt clauses [5]. This measure simply counts the number of distinct decision levels of literals occurring in learnt clauses, at the time of their creation. Thanks to that, new strategies for deleting clauses were proposed. Moreover, the solver constantly watches the quality of the last learnt clauses and triggers a restart when the quality is worst than a dynamic threshold. Recent developments include a way of postponing restarts when the number of assigned literals suddenly increases without conflicts (a SAT solution may be then expected). In the last version of Glucose, a special policy allows the solver to decide which strategy to use with respect to a set of identified extreme case [6].

Indeed, learnt clauses removal, restarts, small modifications of the VSIDS heuristic are based on the concept of LBD. The core engine of Glucose (and Syrup) is 9 years old. Syrup is a parallel version of Glucose on which we focused most of our efforts in the last years, but there is no parallel track this year.

Glucose and Syrup were not well ranked in the 2018 SAT competition. We partially explain in this short description why we choose to keep Glucose untouched despite this.

II. Components introduced in 2018

The 2018 version of Glucose and Syrup are very similar to the 2016 ones, with two improvements. The main modifications are based on the extension of the recent LCM strategies proposed recently [7] (which "revived" the vivification technique [8]). We observed that the LCM strategy was not always performed on clauses of small LBD only, because LCM was not triggered right after clause database reduction, and thus the order of clauses traversed by the LCM was not based on a sorted order of learnt clauses. However, we observed that LCM was more efficient when not always run on good clauses only (LCM can replace clauses, and thus may delete a good clause). We observed that LCM was more efficient when active clauses were kept, in addition to clauses of small LBD. Glucose is now keeping 10% of the most active clauses in addition to the usual LBD based ranking.

III. Components in 2019

Despite the rapid adoption by the community of all the MapleLCMDist mechanisms, we only incorporated the LCM technology, as stated above.

Following the average ranking of Glucose in the 2018 SAT Competition, we decided to test more formally the performances of Glucose vs MapleLCMDistChronoBT [9], the clear winner of the last year contest. We gathered all the available problems and partitioned them by year. We obtained around 3800 problems on which we ran Glucose 4.2.1 (version participating in the 2018 competition) and MapleLCMDistChronoBT. Results are shown in Figure 1. We observed that, despite the very good results in the last competitions, the progress made by the recent solvers is more difficult to measure on previously used problems. It shows that technologies developed in Glucose in the last years are more efficient on problems before 2013. At the opposite, MapleLCMDistChronoBT is the method of choice for the recent sets of problems. More importantly, it seems that all the technologies proposed in the last years (except LCM) are hurting the performances of SAT solvers on problems before 2013. Our intuition on this results is that the MapleLCMDistChronoBT solver is more suited to problems containing arithmetic constraints (typically classified as "Crafted" problems in the previous competitions) while Glucose is more suited to "Industrial" problems, as typically gathered in the previous competitions.

We thus decided to maintain Glucose as is.

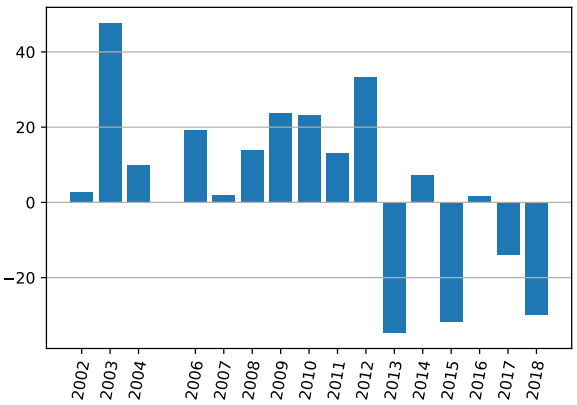


Fig. 1. Percentage of problems on which Glucose is significantly better than MaplLCMDistChronoBT, grouped by year of benchmark used in competitions, Industrial category (when applicable).The X axis is the percentage of all problems, introduced or used for the given year, where Glucose outperforms MapleLCMDistChronoBT (both versions from 2018). Only not easy problems are considered here (not solved by Glucose in less than 100,000 conflicts). We say that a solver outperforms another one if it can solve a problem that is unsolved by the other one (time out of 45 minutes), or if it can solve a problem at least five times more faster than the other.

IV. Algorithm and Implementation details

Glucose uses a special data structure for binary clauses, and a very limited self-subsumption reduction with binary clauses, when the learnt clause is of interesting LBD. The certified UNSAT version of Glucose is using text-based logging informations of the proof.

V. Acknowledgments

Recent developments of Glucose and Syrup are supported by the French Project SATAS ANR-15-CE40-0017.

References

[1] G. Audemard and L. Simon, "On the glucose sat solver," vol. 27, no. 01.

[2] ——, "Glucose: a solver that predicts learnt clauses quality," *SAT Competition*, pp. 7–8, 2009.

[3] ——, "Glucose 2.3 in the sat 2013 competition," *Proceedings of SAT Competition*, pp. 42–43, 2013.

[4] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, 2003, pp. 502–518.

[5] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *IJCAI*, 2009.

[6] ——, "Extreme Cases in SAT," in *19th International Conference on Theory and Applications of Satisfiability Testing(SAT'13)*, 2013, p. To appear.

[7] M. Luo, C. Li, F. Xiao, F. Manyà, and Z. Lü, "An effective learnt clause minimization approach for CDCL SAT solvers," in *Proceedings of IJCAI 2017*, 2017, pp. 703–711.

[8] C. Piette, Y. Hamadi, and L. Sais, "Vivifying propositional clausal formulae," in *ECAI 2008 - 18th European Conference on Artificial Intelligence*, 2008, pp. 525–529.

[9] V. Ryvchin and A. Nadel, "Maple_LCM_Dist_ChronoBT: Featuring chronological backtracking," in *Proceedings of SAT Competition*, 2018.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

Maple_LCM_BTL, MapleLCMChronoBT_ldcr, Glucose_BTL and Glucose_421_DEL

1st Yang Xu, 4th Shuwei Chen
School of Mathematics
National-Local Joint Engineering Laboratory of System Credibility Automatic Verification, Southwest Jiaotong University
Chengdu, China
xuyang@home.swjtu.edu.cn, swchen@home.swjtu.edu.cn

2nd Wenjing Chang, 3rd Xiulan Chen
School of Information Science and Technology
National-Local Joint Engineering Laboratory of System Credibility Automatic Verification, Southwest Jiaotong University
Chengdu, China
wenjing1021@163.com, 534918591@qq.com

***Abstract*— This document describes Maple_LCM_BTL and its friends.**

I. INTRODUCTION

The improvements of these solvers are made mainly from the following two aspects. First, we improved the evaluation method of VSIDS, so that better branch decision variables can be selected. The second improvement is on the evaluation method of learnt clauses so that better clauses can be preserved.

II. IMPROVEMENT

Variable state independent decaying sum (VSIDS) [1] is still the dominant branching heuristics because of its low cost. VSIDS consists of a rewarding mechanism for variables participating in the conflict. We proposed a method to reward variables differently depending on information provided by the conflict analysis process, i.e., the literal block distance value of the learnt clause and the size of the backtrack level decided by the learnt clause [2]. We implement it as part of Glucose 4.1 and Maple_LCM_Dist, and the corresponding solvers are named as Glucose_BTL and Maple_LCM_BTL.

Two major problems need to be solved for the deletion of learnt clauses, which clauses to be deleted and when to delete. Most state of art SAT solvers use LBD value as the evaluation index and delete clauses according to different LBD thresholds.

However, the differences of the learnt clauses with the same LBD value are not considered. Only the first 50% of the learnt clauses in the intermediate sequence and with the same LBD value will be deleted. Some clauses will be deleted even though they might be useful for the future.

We therefore defined a new evaluation value—LD（Level Distance）that measures the difference of the max decision level of a literal ‘v’ in the learnt clause ‘c’ and its LBD value.

$$LD(c) = \max(decisionlevel(v)) - LBD(c) \quad (1)$$

When the LBD values of some learnt clauses are the same, the solver will retain the learnt clause with smaller LD. It means that we will keep the learnt clauses that are on top level. In order to implement this function we resort the learnt clauses in descending order when they have the same LBD value. This method is called LDCR(Level Distance Based Clause Reduction Heuristic).

MapleLCMChronoBT_ldcr is based on MapleLCMDistChronoBT [3] and add the LDCR method.

Glucose_421_DEL is a SAT solver that incorporate the deletion strategy in Maple_LCM_Scavel [4].

REFERENCES

[1] Moskewiez M., Madigan C., Chaff: engineering an efficient SAT solver, in Proceeding of 38th ACM/IEEE Design Automation Conference, Las Vegas, 2001, pp. 530–538.

[2] Chang W., Xu Y., Chen S.. A new rewarding mechanism for branching heuristic in SAT solvers. International Journal of Computational Intelligence Systems, 2019, 12(1). DOI: 10.2991/ijcis.2019.125905649.

[3] Vadim R. and Nadel A. Maple_LCM_Dist_ChronoBT: Featuring Chronological Backtracking. SAT COMPETITION 2018, 29.

[4] Xu Y., Wu G., Chen Q. and Chen S.. Maple_LCM_Scavel and Maple_LCM_Scavel 200. SAT COMPETITION 2018, 30.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

MapleCOMSPS_LRB_VSIDS and MapleCOMSPS_CHB_VSIDS in the 2019 Competition

Jia Hui Liang[*‡], Chanseok Oh[†‡], Krzysztof Czarnecki[*], Pascal Poupart[*], Vijay Ganesh[*]
[*] University of Waterloo, Waterloo, Canada
[†] Google, New York, United States
[‡] Joint first authors

***Abstract*—This document describes the SAT solvers MapleCOMSPS_LRB_VSIDS and MapleCOMSPS_CHB_VSIDS that implement our machine learning branching heuristics called the *learning rate branching heuristic* (LRB) and the *conflict history-based branching heuristic* (CHB).**

I. Introduction

A good branching heuristic is vital to the performance of a SAT solver. Glancing at the results of the previous competitions, it is clear that the VSIDS branching heuristic is the de facto branching heuristic among the top performing solvers. We are submitting two unique solvers with a new branching heuristic called the *learning rate branching heuristic* (LRB) [1] and another solver with the *conflict history-based branching heuristic* (CHB) [2].

Our intuition is that SAT solvers need to prune the search space as quickly as possible, or more specifically, learn a high quantity of high quality learnt clauses. In this perspective, branching heuristics can be viewed as a bi-objective problem to select the branching variables that will simultaneously maximize both the quantity and quality of the learnt clauses generated. To simplify the optimization, we assumed that the first-UIP clause learning scheme will generate good quality learnt clauses. Thus we reduced the two objectives down to just one, that is, we attempt to maximize the quantity of learnt clauses.

II. Learning Rate Branching

We define a concept called *learning rate* to measure the quantity of learnt clauses generated by each variable. The learning rate is defined as the following conditional probability, see our SAT 2016 paper for a detailed description [1].

$$learningRate(x) = \mathbb{P}(Participates(x) \mid Assigned(x) \wedge SolverInConflict)$$

If the learning rate of every variable was known, then the branching heuristic should branch on the variable with the highest learning rate. The learning rate is too difficult and too expensive to compute at each branching, so we cheaply estimate the learning rate using multi-armed bandits, a special class of reinforcement learning. Essentially, we observe the number of learnt clauses each variable participates in generating, under the condition that the variable is assigned and the solver is in conflict. These observations are averaged using an exponential moving average to estimate the current learning rate of each variable. This is implemented using the well-known *exponential recency weighted average algorithm* for multi-armed bandits [3] with learning rate as the reward.

Lastly, we extended the algorithm with two new ideas. The first extension is to encourage branching on variables that occur frequently on the reason side of the conflict analysis and adjacent to the learnt clause during conflict analysis. The second extension is to encourage locality of the branching heuristic [4] by decaying unplayed arms, similar to the decay reinforcement model [5], [6]. We call the final branching heuristic with these two extensions the *learning rate branching heuristic*.

III. Conflict History-Based Branching

The *conflict history-based branching heuristic* (CHB) precedes our LRB work. CHB also applies the exponential recency weighted average algorithm where the reward is the reciprocal of the number of conflicts since the assigned variable last participated in generating a learnt clause. See our paper for more details [2].

IV. Solvers

All the solvers are modifications of COMiniSatPS [7]. We used the same COMiniSatPS version that also participates in the competition [8]. This year's solvers are identical to the last year's solvers.

V. Availability and License

Source is available for download for all the versions described in this paper. All the solvers use the same license as COMiniSatPS. Note that the license of the M4RI library (which COMiniSatPS uses to implement Gaussian elimination) is GPLv2+.

Acknowledgment

We thank specifically the authors of Glucose, GlueMiniSat, Lingeling, CryptoMiniSat, and MiniSat.

REFERENCES

[1] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning Rate Based Branching Heuristic for SAT Solvers," in *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing*, ser. SAT'16, 2016.

[2] ——, "Exponential Recency Weighted Average Branching Heuristic for SAT Solvers," in *Proceedings of AAAI-16*, 2016.

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[4] J. H. Liang, V. Ganesh, E. Zulkoski, A. Zaman, and K. Czarnecki, "Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers," in *Hardware and Software: Verification and Testing*. Springer, 2015, pp. 225–241.

[5] I. Erev and A. E. Roth, "Predicting How People Play Games: Reinforcement Learning in Experimental Games with Unique, Mixed Strategy Equilibria," *American Economic Review*, vol. 88, no. 4, pp. 848–881, 1998.

[6] E. Yechiam and J. R. Busemeyer, "Comparison of basic assumptions embedded in learning models for experience-based decision making," *Psychonomic Bulletin & Review*, vol. 12, no. 3, pp. 387–402.

[7] C. Oh, "Improving SAT Solvers by Exploiting Empirical Characteristics of CDCL," Ph.D. dissertation, New York University, 2016.

[8] ——, "COMiniSatPS Pulsar and GHackCOMSPS in the 2019 Competition," in *SAT Race*, 2019.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

MapleLCMDistChronoBT-DL, duplicate learnts heuristic-aided solvers at the SAT Race 2019

Stepan Kochemazov *, Oleg Zaikin†, Victor Kondratiev ‡ and Alexander Semenov§
Laboratory 6.2, ISDCT SB RAS
Irkutsk, Russia
Email: *veinamond@gmail.com, †zaikin.icc@gmail.com, ‡vikseko@gmail.com, §biclop.rambler@yandex.ru

***Abstract*—This document describes the `MapleLCMDistChronoBT-DL` family of solvers which are based on the SAT Competition 2018 winner, the `MapleLCMDistChronoBT` solver, augmented with duplicate learnts heuristic.**

I. Duplicate Learnts

During the CDCL inference, some learnt clauses can be generated multiple times. It is reasonable to assume that they deserve special attention. In particular, the simple rule for their processing can look as follows: if a learnt clause was repeated at least k times ($k \geq 2$) during the derivation, then this clause should be permanently added to the conflict database. It can be naturally implemented for solvers based on `COMiniSatPS` [1], since they store learnt clauses in three tiers: *core*, *tier2* and *local*, where the learnts in *core* are not subject for reduceDB-like procedures. Thus we basically can put duplicate learnts into *core* when they satisfy the conditions outlined below.

In the submitted solvers we track the appearances of duplicate learnts using a hashtable-like data structure and process them based on several parameters. The hashtable is implemented on top of C++ 11 `unordered_map` associative container. The goal of parameters is to ensure that the hashtable does not eat too much memory, that the learnt clauses are filtered based on their LBD and that the learnts repeated a prespecified number of times are added to *tier2*/*core*.

- `lbd-limit` – only learnt clauses with `lbd` $\leq$ `lbd-limit` are screened for duplicates.
- `min-dup-app` – learnt clauses that repeated `min-dup-app` times are put into *tier2*, and the ones repeated `min-dup-app`+1 times – to *core* tier.
- `dupdb-init` – the initial maximal number of entries in the duplicate learnts hashtable.

The duplicates database is purged as soon as its size exceeds `dupdb-init`. Only the entries corresponding to learnt clauses repeated at least `min-dup-app` times are preserved.

Additionally, we limit `core_lbd_cut` parameter of the solver to 2 since duplicate learnts can provide a lot of additional clauses to store in *core*.

All `MapleLCMDistChronoBT-DL` solvers are based on the SAT Competition 2018 main track winner, `MapleLCMDistChronoBT` [2], which in turn is based on `Maple_LCM_Dist` [3].

The research was funded by Russian Science Foundation (project No. 16-11-10046).

II. MapleLCMDistChronoBT-DL-v1.1

The solver employs `lbd-limit`=12, `min-dup-app`=3 (e.g. only learnts repeated 4 times are added to *core*), and `dupdb-init`=500000. In addition to `lbd-limit` it also filters out the learnt clauses which contain > 200 literals. Also, if during the first 500 000 conflicts no duplicate learnt was added to *core* tier, then it switches off the processing of duplicate learnts altogether.

III. MapleLCMDistChronoBT-DL-v2.1

The solver employs `lbd-limit`=12, `min-dup-app`=3 (e.g. only learnts repeated 4 times are added to *core*), and `dupdb-init`=500000.

IV. MapleLCMDistChronoBT-DL-v2.2

This solver employs `lbd-limit`=14, `min-dup-app`=2 (e.g. only learnts repeated 3 times are added to *core*), and `dupdb-init`=1000000. In addition to that, the lbd value separating *tier2* from *local* tier is increased from 6 to 7.

V. MapleLCMDistChronoBT-DL-v3

This solver uses the same parameters as `MapleLCMDistChronoBT-DL-v2.1` but has a deterministic LRB-VSIDS switching strategy: it starts from LRB and switches each time the number of propagations since the last switch exceeds a specific value. This value starts from 30 000 000 propagations and is increased by 10% each switch.

References

[1] C. Oh, "Between SAT and UNSAT: The fundamental difference in cdcl SAT," in *SAT*, ser. LNCS, vol. 9340, 2015, pp. 307–323.
[2] A. Nadel and V. Ryvchin, "Chronological backtracking," in *SAT*, ser. LNCS, vol. 10929, 2018, pp. 111–121.
[3] M. Luo, C. Li, F. Xiao, F. Manyà, and Z. Lü, "An effective learnt clause minimization approach for CDCL SAT solvers," in *IJCAI*, 2017, pp. 703–711.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

Improving Maple_LCM_Dist_ChronoBT via Variable Reindexing Invariance

Chris Cameron, Xing Jin*, and Kevin Leyton-Brown
University of British Columbia, Wuhan University*
{cchris13, kevinlb}@cs.ubc.ca, deimos.xing@outlook.com*

Abstract

Variable indices imply the initial variable ordering in many SAT solvers. Our submission strategically rearranges the variable indices of SAT instances and runs the top ranked solver from the 2018 SAT Competition, Maple_LCM_Dist_ChronoBT. The key idea is to map the static structure of a SAT problem to a permutation over variable indices, with the goal that related variables should have related indices.

1 Description

Modern SAT solvers sometimes benefit from the order in which a problem encoding is written down; hence, they sometimes see performance degrade under random variable renamings, despite the fact that each corresponds to a logically identical formula [Heule *et al.*, 2019]. We introduce a SAT preprocessor that makes solvers invariant to isomorphic reorderings by mapping the static structure of a CNF SAT problem to the permutations over variable indices. Our system also requires post processing to remap the variable indices for verifying SAT assignments and DRAT proofs. The primary effect of variable renaming is in the initial variable ordering in many SAT solvers. We created a large design space of variable index remapping strategies motivated by work relating static structure to modern branching heuristics (e.g., community structure and VSIDS [Newsham *et al.*, 2015]). After variable reindexing, we use the Maple_LCM_Dist_ChronoBT solver, which was the top ranked solver at the 2018 SAT competition [Ryvchin and Nadel, 2018][Luo *et al.*, 2017]. We optimized for PAR2 performance over the past three main tracks of the SAT competition.

The remapping strategy that we selected traverses the graphical representation of CNF SAT problems using information related to vertex degrees and community structure. First, our preprocessor creates a weighted variable incident graph. Each variable represents a vertex. The weight of an edge between two vertices is the number of clauses that the corresponding variables jointly participate in. Second, we cluster the graph based on community structure using a greedy modularity maximization algorithm. Next, we order the communities by decreasing number of vertices and traverse each community in a breadth first search beginning with the highest degree vertex in each community. The traversal order over vertices represents the new variable indices.

The key idea of our preprocessor is to encourage related variables to have related indices. Because variable indices represent the initial variable ordering, related variables will initially be selected close together. We speculate that this might lead to fast clause learning early on in solver execution where related variables trigger fast contradictions if selected in quick succession. This may have connections to the powerful variable ordering heuristic *learning rate branching* (LRB), where variables are prioritized for assignment if they tend to promote learning of new clauses [Liang *et al.*, 2016].

Computing the community structure is prohibitively expensive for large SAT problems. We only run our preprocessor for instances where the community structure can be efficiently computed. We build a random forest prediction model based on simple size information to predict whether community structure can be computed within 100 seconds. If our model predicts that graph computation will be too expensive or we do the computation and it takes more than 100 seconds, then we run Maple_LCM_Dist_ChronoBT with the original variable indices. Based on the instances from the past three competitions, we expect that we will use our preprocessor for $\approx 60\%$ of instances.

2 Acknowledgements

We thank the authors of Maple_LCM, Maple_LCM_Dist, and Maple_LCM_Dist_ChronoBT.

References

[Heule *et al.*, 2019] Marijn JH Heule, Benjamin Kiesl, and Armin Biere. Encoding redundancy for satisfaction-driven clause learning. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–58. Springer, 2019.

[Liang *et al.*, 2016] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 123–140. Springer, 2016.

[Luo *et al.*, 2017] Mao Luo, Chu-Min Li, Fan Xiao, Felip Manya, and Zhipeng Lü. An effective learnt clause minimization approach for cdcl sat solvers. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 703–711. AAAI Press, 2017.

[Newsham *et al.*, 2015] Zack Newsham, William Lindsay, Vijay Ganesh, Jia Hui Liang, Sebastian Fischmeister, and Krzysztof Czarnecki. Satgraf: Visualizing the evolution of sat formula structure in solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 62–70. Springer, 2015.

[Ryvchin and Nadel, 2018] Vadim Ryvchin and Alexander Nadel. Maple_lcm_dist_chronobt: Featuring chronological backtracking. *SAT COMPETITION 2018*, page 29, 2018.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

Maple_LCM_OnlineDel

Sima Jamali
Simon Fraser University
Vancouver, Canada
sima_jamali@sfu.ca

David Mitchell
Simon Fraser University
Vancouver, Canada
mitchell@cs.sfu.ca

***Abstract*—We provide a brief introduction to our solvers *Maple_LCM_OnlineDel_19a* and *Maple_CM_OnlineDel_19b*. They use a simple online clause reduction scheme, which involves no sorting and was introduced in [10], within *Maple_LCM_Dist_ChronoBT*, the first place solver in the SAT Competition 2018 [1], [2].**

I. Introduction

Maple_LCM_Dist_ChronoBT won the gold medal of the main track of the SAT competition 2018 [3]. Its clause maintenance strategy, which was inherited from COMiniSatPS, involves three stores of learnt clauses: Core, Tier2 and Local [4], [5]. The size of Core and Tier2 are limited by being selective about which clauses are added. The majority of learnt clauses are added to Local. The size of Local is limited by periodically deleting half of the clauses with lowest activities, as in most recent CDCL solvers [4], [6], [7].

II. Online Deletion

The solvers described here replace the delete-half clause deletion strategy by a new deletion scheme called Online Deletion [10]. In Online Deletion, each time the solver derives a new conflict clause we choose a previously learnt clause to replace with it, according to the following simple method.

Clauses of Local are maintained in a circular list L with an index variable i that traverses the list in one direction. The index indicates the current "deletion candidate" L_i at each time. For deletion, we maintain a clause quality measure Q, and a threshold quality value q. When a new clause C is learnt and needs to be stored in Local, we select the next "low quality" clause in the list to be replaced with C. The index i is showing the next candidate L_i. While $Q(L_i) \geq q$, we increment i to "save" clause L_i for one more "round" in the circular list; The first time $Q(L_i) < q$, we replace L_i with C and delete the "old" clause L_i. The size of Local, indicates how long a round is, and the clause quality measure threshold is chosen in a way that there are always sufficiently enough "low-quality" clauses in the list to be deleted [10].

The submitted solvers use a simple quality measure based on counting how many times a clause has been used in conflict analysis since the last time it was considered for deletion. Q is calculated as follows:
Each clause has a quality measure RUL which is an indicator of its recent usage and LBD. RUL is initialized with 0 when a clause is first learnt. Every time a clause C is used in conflict analysis, its RUL is increase by $12/LBD(C)$. In the solvers submitted to this competition, the clause L_i is saved if its RUL is at least 2. ($Q(L_i) = RUL(L_i)$ and $q = 2$). If a clause is saved, its RUL resets to 0.

III. Maple_LCM_OnlineDel_19a

This solver is built based on the winning solver of the SAT competition 2018, Maple_LCM_Dist_ChronoBT [1]. The delete half clause deletion scheme is replaced by Online Deletion as described above. The maximum size of Local is set to 80,000, which indicates the "length" of each round during most of the run. The solvers was first introduced in [10] with minor difference in RUL (replaced $20/LBD(C)$ with $12/LBD(C)$).

IV. Maple_CM_OnlineDel_19b

This solver is the same as *Maple_LCM_OnlineDel_19a* with 2 differences:

1) Size of Local is set to 50,000
2) Unlike Maple_LCM_Dist_ChronoBT that uses the learnt clause minimization introduced in [8]. This solver has further learnt clause minimization as in the solver *Maple_CM_Dist*, which won the third place in the main track of SAT competition 2018 [9].

References

[1] A. Nadel and R. Vadim, "Chronological Backtracking," in Proceedings of SAT, 2018, pp. 111121.
[2] A. Nadel and R. Vadim, "Maple_LCM_Dist_ChronoBT: Featuring Chronological Backtracking," in Proceedings SAT Competition 2018 - Solver and Benchmark Descriptions, 2018, pp. 29.
[3] SAT Competition 2018, *http://sat2018.forsyte.tuwien.ac.at/index.php.*
[4] C. Oh, "Between SAT and UNSAT: The Fundamental Difference in CDCL SAT," in Proceedings of SAT, 2015, pp. 307323.
[5] C. Oh, "Improving SAT solvers by exploiting empirical characteristics of CDCL", Ph.D. dissertation, New York University, 2016.
[6] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, Learning rate based branching heuristic for SAT solvers, in Proceedings of SAT,2016, pp. 123140.
[7] G. Audemard and L. Simon, Predicting learnt clauses quality in modern SAT solvers, in Proceedings of IJCAI, 2009, pp. 399404.
[8] M. Luo, C.-M. Li, F. Xiao, F. Manya, and Z. Lu, An effective learnt clause minimization approach for CDCL SAT solvers, in Proceedings of IJCAI, 2017, pp. 703711.
[9] M. Lou, F. Xiao, C. Li, F. Manya, Z. Lu and Y. Li, "Maple CM, Maple CM Dist, Maple CM ordUIP and Maple CM ordUIP+ ," in Proceedings SAT Competition 2018 - Solver and Benchmark Descriptions, 2018, pp. 4446.
[10] S. Jamali and D. Mitchell, "Simplifying CDCL Clause Database Reduction," in Proceedings of SAT, 2019.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

MapleLCMChronoBT_Scavel_EWMA and Friends Participating SAT Race 2019

1st Zhihui Li, 2nd Guanfeng Wu
School of Information Science and Technology
National-Local Joint Engineering Laboratory of System Credibility Automatic Verification, Southwest Jiaotong University
Chengdu, China
lizhihui@swjtu.edu.cn, wl520gx@gmail.com

4th Yang Xu, 3rd Qingshan Chen
School of Mathematics
National-Local Joint Engineering Laboratory of System Credibility Automatic Verification, Southwest Jiaotong University
Chengdu, China
xuyang@swjtu.edu.cn,qschen@ swjtu.edu.cn

***Abstract*— This document describes the SAT solvers which used EWMA for the SAT Race 2019.**

I. INTRODUCTION

MapleLCMChronoBT_Scavel_EWMA is an improved SAT solver based on MapleLCMChronoBT and Maple_LCM_Scavel [1, 5]. The improvement is made mainly from the following aspect: We add a new evaluation method which named CTS (cumulative trend strength) [4], which is for evaluating learnt clauses by Exponential Weighted Moving Average (EWMA) [2] so that better clauses can be preserved.

II. IMPROVEMENT

We retain the framework of MapleLCMChronoBT to implement the corresponding sequence backtracking [3]. We also incorporate our technical expertise in Maple_LCM_Scavel, which include two aspects: dynamic comprehensive variable activity evaluation and learnt clause threshold value management strategy. Our improvement work is as follows:

CHEN et al. [4] studied the time series distribution of learnt clauses used and proposed an algorithm for quantifying the cumulative trend strength (CTS) of learning clauses. The random and scattered time distribution is transformed into continuous cumulative trend strength, and the trend strength threshold is set to determine whether the clause should be deleted.

Setting the threshold of cumulative trend strength for learnt clause evaluation is a violent truncation method. In statistics, a moving average is the principle of evaluating a certain window of recent data points related to time series data. It is, e.g., often used in technical analysis of financial data in connection with stock prices. In a similar way, the tendency of learning clauses to participate in conflict analysis can be judged by comparing the short-term and long-term cumulative trend strength.

In order to make better use of the dynamic information of the cumulative trend strength, we use the EWMA algorithm to improve the evaluation criteria of learnt clauses.

Expecting to achieve better performance, we further refined and adjusted relevant operating parameters through experiments, based on our Maple_LCM_Scavel solver of 2018.

MapleLCMDChronoBT_DEL is also based on MapleLCMChronoBT and Maple_LCM_Scavel [1, 5]. It adopts only the deletion strategy in Maple_LCM_Scavel.

REFERENCES

[1] SAT Competition 2018 homepage .http://sat2018.forsyte.tuwien.ac.at/

[2] Biere A., Fröhlich A.. (2015) Evaluating CDCL Variable Scoring Schemes. In: Heule M., Weaver S. (eds) Theory and Applications of Satisfiability Testing -- SAT 2015. SAT 2015. Lecture Notes in Computer Science, vol 9340. Springer, Cham.

[3] Nadel A., Ryvchin V.. Chronological Backtracking. Theory and Applications of Satisfiability Testing-SAT 2018-21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Proceedings

[4] Chen Q., Xu Y., Wu G.. Learn Claus Evaluation Algorithm of SAT Problem Based on Trend Strength. Computer Science, Vol.45 No,12Dec.2018: 137-141

[5] Simon L., Audemard G.. Predicting learnt clauses quality in modern sat solver [C]. In Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09). Menlo Park: AAAI Press, 2009: 399-404.

[6] Minisat homepage [EB/OL]. http://minisat.se/MiniSat.html.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

MergeSAT

Norbert Manthey
nmanthey@conp-solutions.com
Dresden, Germany

***Abstract*—The sequential SAT solver MERGESAT starts from last years competition winner, and adds known as well as novel implementation and search improvements. MERGESAT is setup to simplify merging solver contributions into one solver, to motivate more collaboration among solver developers.**

I. INTRODUCTION

When looking at recent SAT competitions, the winner of the current year was typically last years winner plus a single modification. However, each year there are several novel ideas that the winner does not pick up. Hence, lots of potential with respect to maximal performance is likely lost, and bug fixes of previous versions do not make it into novel ones.

The CDCL solver MERGESAT is based on the competition winner of 2018, MAPLE_LCM_DIST_CHRONOBT [12], and adds several known techniques, fixes, as well as adds some novel ideas. To make continuing the long list of work2 that influenced MERGESAT simpler, MERGESAT uses git to combine changes, and comes with continuous integration to simplify extending the solver further.

II. INTEGRATED TECHNIQUES AND FIXES

Most recently, backtracking has been improved by [8]. Backtracking improvements during restarts have already been proposed in [11]. MERGESAT uses the *partial trail* based backtracking during restarts.

Learned clause minimization (LCM) [6] is also kept. It is still open research in which order literals should be considered during vivification [9]. MERGESAT uses the improvement from [7], which repeats vivification in reverse order, in case a clause could be simplified with the first order. The original implementation of LCM adds a bit to the clause header to indicate that this clause has been considered already. However, no other bit has been dropped from the header, resulting in a 65 bit header structure. Along [4], this can result in a major slow down of the solver. Consequently, MERGESAT drops a bit in the size representation of the clause.

Large formulas require a long simplification time, even though simplification algorithms are polynomial. While for a 5000 second timeout, large simplification times are acceptable for effective simplifications, usually an incomplete simplification helps the solver already. Therefore, we introduce a step counter, that is increased whenever simplification touches a clause. Next, we interrupt simplification as soon as this counter reaches a predefined limit, similarly to [2]. To speed simplification up further, the linear subsumption implementation and related optimizations from [3] have been integrated into MERGESAT.

Since the solver MAPLESAT [5], the decision heuristic is switched back from the currently selected one to VSIDS – after 2500 seconds. As solver execution does not correlate with run time, this decision results in solver runs not being reproducible. To fix this property, the switch to VSIDS is now dependent on the number of performed propagations as well as conflicts. Once, one of the two hits a predefined limit, the heuristic is switched back to VSIDS. This change enables reproducibility and deterministic behavior again.

MERGESAT implements an experimental – and hence disabled by default – heuristic to decide when to disable *phase saving* [10] during backtracking, which has been used in RISS [7] before: When the formula is parsed, for each non-unit clause it is tracked whether before applying unit propagation there is a positive literal. In case there is no positive literal, a *break* count is incremented. For the whole formula, this count approximates how close the formula is to being able to be solved by the *pure literal rule*. In case this break count is zero, or below a user defined threshold, no phase saving is used. The same rule is applied for negative literals. There exists benchmarks, where this heuristic with a threshold zero results in a much better performance. However, for a mixed benchmark, this feature has not been tested enough, and hence, remains disabled.

III. INCREMENTAL SAT

In previous variants of MAPLESAT, incremental solving via assumptions was disabled. To be able to use MERGESAT as backend in model checkers and other tools that require incremental solving capabilities, this feature is brought back. Furthermore, an extended version [1] of the IPASIR interface [1] is provided, which besides the usual functionality offers an additional function *ipasir_solve_final* to tell the SAT solver that this call is the final (or only) call. This function allows the solver to use formula simplification more extensively, because usually simplification cannot be applied during incremental solving.

IV. AVAILABILITY

The source of the solver is publicly available under the MIT license at https://github.com/conp-solutions/mergesat. The version with the git tag "satrace-2019" is used for the submission. The submitted starexec package can be reproduced by running "./scripts/make-starexec.sh" on this commit.

[1] https://github.com/conp-solutions/ipasir

V. Continuous Testing

The submitted version of MergeSAT compiles on Linux and Mac OS. GitHub allows to use continuous testing, which essentially build MergeSAT, and tests basic functionality: i) producing unsatisfiability proofs, ii) building the starexec package and producing proofs, iii) being used as an incremental SAT backend in Open-WBO as well as iv) solving via the IPASIR interface. All these steps are executed by executing the script "tools/ci.sh" from the repository, and the script can be used as a template to derive similar functionality.

Acknowledgment

The author would like to thank the developers of all predecessors of MergeSAT, and all the authors who contributed the modifications that have been integrated.

References

[1] T. Balyo, A. Biere, M. Iser, and C. Sinz, "SAT race 2015," *Artif. Intell.*, vol. 241, pp. 45–65, 2016.

[2] A. Biere, "PrecoSAT system description," http://fmv.jku.at/precosat/preicosat-sc09.pdf, 2009.

[3] T. Ehlers and D. Nowotka, "Tuning parallel sat solvers," in *Proceedings of Pragmatics of SAT 2015 and 2018*, ser. EPiC Series in Computing, D. L. Berre and M. J\"arvisalo, Eds., vol. 59. EasyChair, 2019, pp. 127–143. [Online]. Available: https://easychair.org/publications/paper/NkG7

[4] S. Hölldobler, N. Manthey, and A. Saptawijaya, "Improving resource-unaware SAT solvers," ser. LNCS, C. G. Fermüller and A. Voronkov, Eds., vol. 6397. Heidelberg: Springer, 2010, pp. 519–534.

[5] V. G. K. C. Jia Hui Liang, Chanseok Oh and P. Poupart, "MapleCOMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB," in *Proceedings of SAT Competition 2016*. [Online]. Available: http://hdl.handle.net/10138/164630

[6] M. Luo, C.-M. Li, F. Xiao, F. Manyà, and Z. Lü, "An effective learnt clause minimization approach for cdcl sat solvers," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017, pp. 703–711. [Online]. Available: https://doi.org/10.24963/ijcai.2017/98

[7] N. Manthey, "Riss 7.1," in *Proceedings of SAT Competition 2018*. [Online]. Available: http://hdl.handle.net/10138/237063

[8] A. Nadel and V. Ryvchin, "Chronological backtracking," in *Theory and Applications of Satisfiability Testing – SAT 2018*, O. Beyersdorff and C. M. Wintersteiger, Eds. Cham: Springer International Publishing, 2018, pp. 111–121.

[9] C. Piette, Y. Hamadi, and L. Sais, "Vivifying propositional clausal formulae," in *ECAI*, ser. Frontiers in Artificial Intelligence and Applications, vol. 178, 2008, pp. 525–529.

[10] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *SAT*, ser. LNCS, vol. 4501, 2007, pp. 294–299.

[11] A. Ramos, P. van der Tak, and M. J. H. Heule, "Between restarts and backjumps," in *Theory and Applications of Satisfiability Testing - SAT 2011*, K. A. Sakallah and L. Simon, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 216–229.

[12] V. Ryvchin and A. Nadel, "Maple_LCM_Dist_ChronoBT: Featuring Chronological Backtracking," in *Proceedings of SAT Competition 2018*. [Online]. Available: http://hdl.handle.net/10138/237063

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

Smallsat, Optsat and MapleLCMdistCBTcoreFirst: Containing Core First Unit Propagation

Jingchao Chen
School of Informatics, Donghua University
2999 North Renmin Road, Songjiang District, Shanghai 201620, P. R. China
chen-jc@dhu.edu.cn

***Abstract*—This document uses a new Unit Propagation (called also Boolean Constraint Propagation) heuristic to improve CDCL (conflict-driven clause-learning) SAT solvers. This new heuristic is called Core First Unit Propagation. Generally speaking, the unit propagation is implemented by scanning sequentially every clause over a linear watch-list. Instead of it, we prefer core clauses over non-core ones during unit propagation. Using the core first unit propagation technique, we improve two CDCL SAT solvers: smallsat and MapleLCMDistChronoBT, and design a new SAT solver called optsat**

I. Introduction

Unit propagation (which is called also Boolean Constraint Propagation) is not only an important component of every modern CDCL SAT solver, but also an important one of some proof checkers [1], [2]. BCP (Boolean Constraint Propagation) carries out repeatedly the identification of unit clauses and the creation of the associated implications until either no more implications are found or a conflict (empty clause) is produced. In general, BCP is implemented by scanning sequentially every clause over a linear watch-list. Based on our empirical observation, we found that this implementation is not efficient in some cases. Because of this, this article proposes a new unit propagation heuristic called core first unit propagation. Its basic idea is to prefer core clauses over non-core ones during unit propagation, trying to generate a shorter learnt clause.

II. Core First Unit Propagation

In this article, a clause is said to be core if it is a learnt clause and its LBD (Literal Block Distance) value is less than 7, where LBD is defined as the number of decision variables in a clause [5]. Here the core concept corresponds to the concept of non local in the CoMiniSatPS solver that classifies learnt clauses into three categories [6]. It is different from the core concept used in [1], [2], where core clauses refer to marked or visited ones, and have nothing to do with LBD. The basic idea of CFUP (Core First Unit Propagation) is to prefer core clauses over non-core ones during unit propagation. This can be done by moving core clauses ahead of non-core clauses. The pseudo-code of CFUP shown in Algorithm 2 assumes that a full literal watch scheme (a full occurrence list of all clauses) is used, If using a two literal watch scheme [7], The statement "Append $W[l] - C$ to the end of C " in Algorithm 2 can be replaced with the code of Algorithm 1.

A general CDCL solver has two watchlists: binary and non binary. We adopt the core priority strategy only on a non-binary watchlist. By our empirical observation, adopting always the core priority strategy is not good choice. A better policy is that when the number of conflicts is less than 2×10^6, **CFUP** is called, Otherwise, **BCP** is called. The high-level algorithm CDCL combining **CFUP** and **BCP** are shown in Algorithm 3. CDCL given in Algorithm 3 uses a loop to reach a status where either all the variables are assigned (SAT) or an empty clause is derived (UNSAT). Inside the loop, based on whether the number of conflicts is greater than θ, it decides to invoke either **CFUP** or **BCP**. Here **BCP** is considered a unit propagation without any priority strategy. If there is a conflict, **CFUP** or **BCP** returns a falsified conflicting clause. Otherwise, a new decision is taken and pushed to the trail stack. Conflict analysis learns a new 1UIP clause γ. CDCL asserts the unassigned 1UIP literal and pushes it to the trail stack.

Algorithm 1 Append non-core to core under two literal watch scheme

$W[l]$: set of clauses watched by literal l
  $D := \emptyset$
  for $k = 0$ to end index of $W[l]$ **do**
    if $W[l][k]$ has more than two unassigned literals **then**
      $D := D \cup \{W[l][k]\}$
      $W[\bar{s}] = D \cup \{W[l][k]\}$
      where s is unwatched and unassigned literal
    end if
  end for
  Append $W[l] - C - D$ to the end of C

III. smallsat

Here, smallsat is an improved version of smallsat 2018 [9]. This version adds **CFUP** and adopt the mixed learning rate. The learning rate r_v in this solver is defined as

$$r_v = \frac{C_v + S_v + P_v}{T}$$

where C_v, S_v and P_v is the number of conflict clauses, seen clauses and reason clauses variable v participated in since v is picked or assigned, $T = conflictCounter - pickedTime[v]$ when $LBD < 22$, $T = conflictCounter - assigedTime[v]$ otherwise, where $pickedTime[v]$ and $assigedTime[v]$ are the number of conflicts since v is picked and assigned,

Algorithm 2 CFUP( ): Core First Unit Propagation

T: trail stack of decisions and implications
$W[l]$: set of clauses watched by literal l

```
β := null
for q := index of 1st unvisited literal in T to T.size do
  l := T[q]
  C := ∅, where C is used to store core clauses
  for k = 0 to W[l].size do
    if W[l][k] is unit then
      u := the unassigned literal of W[l][k]
      Push u to the end of T
      if W[l][k] is core clause then
        C := C ∪ {W[l][k]}
      else
        if W[l][k] is falsified then
          β := W[l][k] break
        end if
      end if
    end if
    Append W[l] − C to the end of C
    W[l] := C
    if β ≠ null then return β
  end for
end for
return null
```

Algorithm 3 CDCL(): Conflict-Driven Clause Learning

T: trail stack of decisions and implications
γ: a learnt clause

```
while not all variables assigned do
  if No_of_conflict > θ then
    conflict_cls := BCP()
  else
    conflict_cls := CFUP()
  end if
  if conflicting_cls ≠ null then
    (1uip, γ) := ConflictAnalysis(conflicting_cls)
    if γ = ∅ then
      return UNSAT
    end if
    Push 1uip to T
    Backtrack(current decision level-1)
  else
    Decide and push the decision to T
  end if
end while
return SAT
```

respectively. Like the solver inIDGlucose [10], the initial phase of a decision variable in this solver is based on a weighted literal occurrence count on the original CNF.

IV. OPTSAT

This solver is simpler than smallsat. Compared to the above version of smallsat, optsat removes the tree-based branching solving strategy of smallsat. In this solcer, The learning rate r_v is defined as

$$r_v = \frac{C_v + P_v}{T}$$

where C_v and P_v is the number of conflict clauses and reason clauses v participated in since v is assigned, and T is the interval time that is defined as $T = conflictCounter - assigedTime[v]$. The initial phase of a decision variable in this solver is similar to the solver inIDGlucose [10]. That is, it is set by weighting a literal occurrence in the original CNF.

V. MAPLELCMDISTCBTCOREFIRST

This is a hack version of MapleLCMDistChronoBT [3], [4]. This solver adds only **CFUP**. The remainder keeps unchanged. That is, it is the same as MapleLCMDistChronoBT.

REFERENCES

[1] Lammich P.: Efficient Verified (UN)SAT Certificate Checking, CADE 2017, LNCS 10395, pp. 237–254 (2017)
[2] Chen J.: Fast Verifying Proofs of Propositional Unsatisfiability via Window Shifting, (2018), https://arxiv.org/abs/1611.04838
[3] Nadel, A., Ryvchin, V.: Chronological Backtracking, SAT 2018, LNCS 10929, pp. 111–121 (2018)
[4] Ryvchin, V., Nadel, A.: Maple_LCM_Dist_ChronoBT: Featuring Chronological Backtracking, Proceedings of SAT Competition 2018, p.29
[5] Audemard, G., Simon, L.: predicting learnt clauses quality in modern SAT solvers, IJCAI 2009.
[6] Oh, C.: Between SAT and UNSAT: The fundamental difference in CDCL SAT, SAT 2015.
[7] Zhang, H. : SATO: An efficient propositional prover. 14th International Conference on Automated Deduction (CADE), LNCS 1249, 272–275(1997)
[8] Heule, M., Jarvisalo, M., Suda, M.: SAT competition (2018). http://sat2018. forsyte.tuwien.ac.at/
[9] Chen J.: AbcdSAT and Glucose hack: Various Simplifications and Optimizations for CDCL SAT solvers, Proceedings of SAT Competition 2018, pp.10-12
[10] Devriendt J.: inIDGlucose, Proceedings of SAT Competition 2018, p.41

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

PADC_MapleLCMDistChronoBT, PADC_Maple_LCM_Dist and PSIDS_MapleLCMDistChronoBT in the SR19

Rodrigue Konan Tchinda
Department of Mathematics and Computer Science
University of Dschang
Dschang, Cameroon
rodriguekonanktr@gmail.com

Clémentin Tayou Djamegni
Department of Mathematics and Computer Science
University of Dschang
Dschang, Cameroon
dtayou@gmail.com

***Abstract*—This document describes the solvers PADC_MapleLCMDistChronoBT, PADC_Maple_LCM_Dist and PSIDS_MapleLCMDistChronoBT which integrate the PADC (periodic aggressive learned clause database cleaning) strategy and the PSIDS (Polarity State Independent Decaying Sum) heuristic.**

I. Introduction

The boolean satisfiability problem (SAT) has seen tremendous progresses in its resolution these last years thanks to the integration of several features within the so-called CDCL (Conflict-Driven Clause Learning) [1]–[4] SAT solvers which made them capable of effectively solving several previously intractable instances. These features include clause learning, efficient unit propagation through watched literals, effective learned clause database management, dynamic branching heuristics, restarts etc. SAT solvers nowadays implement many heuristics and are highly sensible to slight modifications in their source codes. That is, a simple change in the source code can result in a solver with completely different performances. A typical example is the great number of solvers that participated in the last SAT competitions, which are the result of very simple modifications of Minisat [5]/Glucose [6], [7] and which showed important performances improvement. The methods we use in this document follows the same idea, i.e. improving performances with slight modifications. The first is the PADC (periodic aggressive learned clause database cleaning) strategy [8] and the second is the PSIDS (Polarity State Independent Decaying Sum) polarity heuristic. They have been integrated into the winners of the last two SAT competitions namely MapleLCMDistChronoBT [9] and Maple_LCM_Dist [10] in order to participate in the 2019 SAT Race. The PADC strategy showed good performances in the 2018 SAT competition when integrated within Glucose-3.0[1] and its current integration into the winners of the previous SAT Competitions revealed significant improvements during preliminary experiments conducted on the latest competitions' benchmarks. We are impatient this year to see how our integration of PADC into MapleLCMDistChronoBT and Maple_LCM_Dist will perform on the new benchmark set. PSIDS (Polarity State Independent Decaying Sum) as far as it is concerned is a polarity heuristic which closely follows the principle of the VSIDS branching heuristic [11].

[1]https://www.labri.fr/perso/lsimon/downloads/softwares/glucose-3.0.tgz

II. PADC

The PADC (periodic aggressive learned clause database cleaning) strategy [8] is an aggressive learned clause database cleaning strategy which periodically deletes a large amount of clauses in the learned clause database. This technique allows the solver to periodically perform a deep cleaning of the learned clause database. Concretely, after every $K-1$ executions of the cleaning procedure (i.e. at the K^{th} execution after the previous deep cleaning step), all the learned clauses are removed, except those of very high quality — such as clauses with $LBD \leq 2$ — and those that are involved in the construction of the implication graph. This aggressive learned clause database reduction has some positive impact on the solver's performances such as increasing diversification, reducing memory consumption and speeding up unit propagations. We integrated this technique within MapleLCMDistChronoBT and Maple_LCM_Dist and called the resulting solvers PADC_MapleLCMDistChronoBT and PADC_Maple_LCM_Dist respectively. Note that these solvers use a three-tiered learned clauses database where the learned clauses are divided into the following three sets : CORE, TIER2 and LOCAL. CORE and TIER2 store clauses with $LBD \leq 6$, the best of which being stored in CORE while LOCAL stores the others. We introduced in them a parameter called *ClearType* in order to control which learned clause database to clear during deep cleaning steps. *ClearType* can take three values : $0, 1$ or 2 indicating respectively to clean the LOCAL learned clause database only, the LOCAL and TIER2 databases and all databases.

III. PSIDS HEURISTIC

Branching and polarity heuristics are known to be determinant for the performances of SAT solvers. Once the branching heuristic has chosen the next variable to branch on, the polarity heuristic comes into play to determine which polarity to set for the latter. The de facto standard branching and polarity heuristics today are VSIDS (Variable State Independent Decaying Sum) [11] and *progress saving* [12] (also known as *phase saving*) respectively. Although there have been several attempts to replace it, VSIDS still remains a widely used branching heuristic in modern SAT solvers. It operates by choosing for branching, the most active unassigned variable in the solver. *Progress saving* as for it was introduced to prevent repeated work in solvers since upon non chronological backtracking there can be some redundant rediscovery of some sub-problems' solutions lost while unassigning variables. The limitation with *progress saving* is that it takes into account only the saved polarities of the assignment preceding the backjumping. This can still lead to the rediscovery of some sub-problems' solutions as some polarities may change after setting the asserting literal. The polarity that is frequently used might be a good choice at this level since it might more likely be the one which make the sub-problem satisfied: this is the intuition behind the PSIDS (Polarity State Independent Decaying Sum) heuristic. The PSIDS heuristic is similar to the VSIDS heuristic but is used for polarities instead of variables. Concretely, we keep for each variable in the solver two scores for its positive and negative polarities respectively. Each time a polarity — of a variable — is set in the solver, the activity of the latter is increased, and when a decision is made using the branching heuristic, then the most active polarity is chosen. As with VSIDS, we decrease from time to time the activities of all polarities (of all variables) in order to favor most recent ones. This heuristic unlike *progress saving* [12] takes into account the agility of the polarity of a variable in a longer period.

IV. SAT RACE 2019 SPECIFICS

We submitted two configurations of PADC_MapleLCMDistChronoBT : the first with parameter K set to 10 and with *ClearType* set to 0, and the second with K set to 5 and *ClearType* set to 0 as well. As far as PADC_Maple_LCM_Dist is concerned, we set its parameter K to 10 and *ClearType* to 0. We also integrated the PSIDS heuristic within MapleLCMDistChronoBT and submitted the resulting solver PSIDS_MapleLCMDistChronoBT to the 2019 SAT Race.

V. ACKNOWLEDGMENTS

Many thanks to the authors of MapleLCMDistChronoBT [9] and Maple_LCM_Dist [10].

REFERENCES

[1] J. Marques-Silva and K. Sakallah, "Grasp-a new search algorithm for satisfiability. iccad," 1996.

[2] H. Zhang, "Sato: An efficient prepositional prover," in *Automated DeductionCADE-14*. Springer, 1997, pp. 272–275.

[3] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver," in *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*. IEEE Press, 2001, pp. 279–285.

[4] C. P. Gomes, B. Selman, H. Kautz *et al.*, "Boosting combinatorial search through randomization," *AAAI/IAAI*, vol. 98, pp. 431–437, 1998.

[5] N. Eén and N. Sörensson, "An extensible sat-solver," in *International conference on theory and applications of satisfiability testing*. Springer, 2003, pp. 502–518.

[6] G. Audemard and L. Simon, "Glucose in the sat 2014 competition," *SAT COMPETITION 2014*, p. 31, 2014.

[7] ——, "On the glucose sat solver," *International Journal on Artificial Intelligence Tools*, vol. 27, no. 01, p. 1840001, 2018.

[8] R. K. Tchinda and C. T. Djamegni, "Scalope, penelope mdlc and glucose-3.0 padc in sc18," *SAT COMPETITION 2018*, p. 42, 2018.

[9] V. Ryvchin and A. Nadel, "Maple_lcm_dist_chronobt: Featuring chronological backtracking," *SAT COMPETITION 2018*, p. 29, 2018.

[10] F. Xiao, M. Luo, C.-M. Li, F. Manya, and Z. Lü, "Maplelrb lcm, maple lcm, maple lcm dist, maplelrb lcmoccrestart and glucose-3.0+ width in sat competition 2017," *Proc. of SAT Competition*, pp. 22–23, 2017.

[11] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th annual Design Automation Conference*. ACM, 2001, pp. 530–535.

[12] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *International conference on theory and applications of satisfiability testing*. Springer, 2007, pp. 294–299.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

Four Relaxed CDCL Solvers

Shaowei Cai and Xindi Zhang
State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences, Beijing, China
shaoweicai.cs@gmail.com

***Abstract*—This note introduce a novel method for improving CDCL-based SAT solvers by relaxing backtrack and integrating local search techniques, and we use this method to improve four state of the art CDCL solvers. The four resulting solvers called Relaxed_LCM_Dist, Relaxed_LCMDistChronoBT, Relaxed_LCMDistChronoBT_p9 and Relaxed_LCMDistChronoBT_Scavel.**

I. Introduction

We propose a method to improve CDCL solvers by relaxing the backtracking. The idea is to relax the backtrack process, by allowing some promising branches to be extended to the leaf (corresponding to a complete assignment) without backtracking, even if conflicts are met during extending the assignment. The resulting complete assignments obtained in this way are highly consistent and are more likely to have small distances to a model (satisfying assignment). Then, a local search solver is called to find a model nearby. If the local search cannot find a model within a short time limit, the CDCL search process continues as normal from the node where the algorithm enters the non-backtracking phase.

II. A Relaxed CDCL Approach

Our method is to relax the backtracking by protecting promising partial assignments from being pruned. Specifically, during the search of CDCL, whenever reaching a node corresponding to a promising assignment, the algorithm enters a non-backtracking phase (with some condition), which uses unit propagation and heuristics in CDCL to assign the remaining variables without backtracking, even an empty clause is detected. At the end, this leads to a complete assignment β,which is fed to a local search solver to search for a model nearby. If the local search fails to find a model within a certain time budget, then the algorithm goes back to the normal CDCL search from the node where it was interrupted (we call this a breakpoint). The non-backtracking phase does not change the data structures used for CDCL search process. So, if all calls of local search fail to find a solution, the modified solver works in the same way just as the original CDCL solver, but it may have a lower speed due to the time consumption of local search.

III. The Local Search Algorithm

As for the local search solver used in our four solvers, we use the CCAnr [1] solver. There are three parameters in the local search solver : the average weight threshold parameter γ, and the two factor parameters ρ and q. All of the three parameters are for the Threshold-based Smoothed Weighting (TSW) weighting scheme.

IV. Main Parameters

There is one parameter p for controlling the cooperation of the backtracking style procedure and the local search solver. Meanwhile, we limit the time of Local search procedure no more than 300 seconds every time ReasonLS call it, and limit the sum of all Local search time no more than a propotion ξ of the total runtime.

For all our relaxed CDCL solvers, the parameters are set as follows with only one exception (p is 0.9 for Relaxed_LCMDistChronoBT_p9).

$p = 0.5$; $\xi = 0.3$; $\gamma = 50$; $\rho = 0.3$; $q = 0.7$.

V. Implementation Details

Our four solvers are implemented in C++. Relaxed_LCM_Dist is developed based on the codes of Maple_LCM_Dist[2] and CCAnr. Relaxed_LCMDistChronoBT and Relaxed_LCMDistChronoBT_p9 are developed based on the codes of Maple_LCM_Dist_ChronoBT[3] and CCAnr, and the only one difference between them is the value parameter p. Besides, Relaxed_LCMDistCB_Scavel uses some techniques from Maple_LCM_Scavel [4] to optimize the performance of ReasonLS_LCMDistCB.

VI. SAT Competition 2019 Specifies

Our four solvers are submitted to "main Track". It is compiled by g++ with the 'O3' optimization option.

They need to be compiled in the root folder by running "./starexec_build".

The running command is: "./starexec_run_default $1" in folder "./bin". The parameter $1 is the absolute path of input file. For a given input file "∼/sc/a.cnf", the call command is "./starexec_run_default ∼/sc/a.cnf ".

References

[1] S. Cai, C. Luo, and K. Su, "CCAnr: A configuration checking based local search solver for non-random satisfiability," in *Proceedings of18th International Conference on Theory and Applications of Satisfiability Testing, SAT 2015, Austin, TX, USA, September 24-27, 2015*, 2015, pp. 1–8.

[2] M. Luo, C. Li, F. Xiao, F. Manyà, and Z. Lü, "An effective learnt clause minimization approach for CDCL SAT solvers," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017, pp. 703–711.

[3] A. Nadel and V. Ryvchin, *Chronological Backtracking*, 06 2018, pp. 111–121.

[4] Y. Xu, G. Wu, Q. Chen, and S. Chen, "Maple_LCM_Scavel and Maple_LCM_Scavel_200," in *Proceeding of SAT Competition 2018 - Solver and Benchmark Descriptions, Department of Computer Science Series of Publications B, University of Helsinki*, 2018, p. 27.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

Riss 7.1 at SAT Race 2019

Norbert Manthey
nmanthey@conp-solutions.com
Dresden, Germany

***Abstract*—The sequential SAT solver RISS combines a heavily modified Minisat-style solving engine of GLUCOSE 2.2 with a state-of-the-art preprocessor COPROCESSOR and adds many modifications to the search process. RISS allows to use inprocessing based on COPROCESSOR. As unsatisfiability proofs are mandatory in 2019, but many simplification techniques cannot produce them, a special configuration is submitted, which first uses all relevant simplification techniques, and in case of unsatisfiability, falls back to the less powerful configuration that supports proofs.**

I. INTRODUCTION

The CDCL solver RISS is a highly configurable SAT solver based on MINISAT [1] and GLUCOSE 2.2 [2], [3], implemented in C++. Many search algorithm extensions have been added, and RISS is equipped with the preprocessor COPROCESSOR [4]. Furthermore, RISS supports automated configuration selection based on CNF formulas features, emitting DRAT proofs for many techniques and comments why proof extensions are made, and incremental solving. The solver is continuously tested for being able to build, correctly solve CNFs with several configurations, and compile against the IPASIR interface. For automated configuration, RISS is also able to emit its parameter specification on a detail level specified by the user. The repository of the solver provides a basic tutorial on how it can be used, and the solver provides parameters that allow to emit detailed information about the executed algorithm in case it is compiled in debug mode (look for "debug" in the help output). While RISS also implements model enumeration, parallel solving, and parallel model enumeration, this document focusses only on the differences to RISS 7, which has been submitted to SAT Competition 2017. Compared to the version of 2018, only the NOUNSAT configuration has been added.

II. SAT COMPETITION SPECIFICS – NOUNSAT CONFIGURATION

The default configuration uses only variable elimination [5] and bounded variable addition [6] as simplification, both of which can produce unsatisfiability proofs.

While recent SAT competitions come with a NOLIMITS track, this years event requires unsatisfiability proofs. To comply, simplification techniques that cannot produce proofs have been disabled in this situation. Differently, this years version comes with the NOUNSAT configuration, which basically cannot produce unsatisfiability answers. This means, that all simplification techniques are available for formulas that are satisfiable, or cannot be solved. In case the formula turns out to be unsatisfiable, the procedure is solved one more time, using the configuration that can produce unsatisfiability proofs.

III. AVAILABILITY

The source of the solver is publicly available under the LGPL v2 license at https://github.com/conp-solutions/riss. The version with the git tag "satrace-2019" is used for the submission. The submitted starexec package can be reproduced by running "./scripts/make-starexec.sh" on this commit.

ACKNOWLEDGMENT

The author would like to thank the developers of GLUCOSE 2.2 and MINISAT 2.2. The development of this project was supported by the DFG grant HO 1294/11-1.

REFERENCES

[1] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT 2003*, ser. LNCS, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Heidelberg: Springer, 2004, pp. 502–518.

[2] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *IJCAI 2009*, C. Boutilier, Ed. Pasadena: Morgan Kaufmann Publishers Inc., 2009, pp. 399–404.

[3] ——, "Refining restarts strategies for sat and unsat," in *CP'12*, 2012, pp. 118–126.

[4] N. Manthey, "Coprocessor 2.0 – a flexible CNF simplifier," in *SAT 2012*, ser. LNCS, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Heidelberg: Springer, 2012, pp. 436–441.

[5] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *SAT 2005*, ser. LNCS, F. Bacchus and T. Walsh, Eds., vol. 3569. Heidelberg: Springer, 2005, pp. 61–75.

[6] N. Manthey, M. J. Heule, and A. Biere, "Automated reencoding of Boolean formulas," in *Hardware and Software: Verification and Testing*, ser. Lecture Notes in Computer Science, A. Biere, A. Nahir, and T. Vos, Eds., vol. 7857. Springer Berlin Heidelberg, 2013, pp. 102–117. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39611-3_14

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

SLIME: A Minimal Heuristic to Boost SAT Solving

Oscar Riveros
Research Architect at www.PEQNP.science
PEQNP
Santiago, Chile.
oscar.riveros@peqnp.science

***Abstract*—This is the system description of the SLIME SAT Solver submitted to the SAT Competition 2019, based in MapleLCMDistChronoBT winner by PAR-2 Score of SAT Competition 2018 [1] on Main Track.**

I. Description

On CDCL Based SAT Solvers the *trail* size is strictly related to progress or to the total conflicts on the current assignment, such that if the *trail* size is the same that the number of variables, then current assignment is valid.

On the other hand, in the selection of the current variable it is necessary to assign a predetermined polarity to the resulting literal, which in most implementations is a predefined value.

SLIME implement a simple heuristic with minimal complexity, that correlated the *trail* size and the polarity of the current variable to assign.

The selection of variable is not related to *trail* size, this decouple the both concepts.

Algorithm 1 Boost: Algorithm for Variable Selection.

Require: variable
Ensure: literal
  polarity[variable] = !polarity[variable]
  if $size(trail) > global$ **then**
    $global = size(trail)$
  else if $size(trail) < global$ **then**
    polarity[variable] = !polarity[variable]
  end if
  return $literal(variable, polarity[variable])$

* The *global* is an external variable init to 0.

II. Experimental Results

Thanks to www.starexec.org it could be possible to test the Boost Heuristics vs The Control Solver, in a real context, these are the results for SAT Race 2006 and SAT Race 2015 at 1800 seconds of wall with the PAR-2 score scheme. (The score of a solver is defined as the sum of all runtimes for solved instances + 2*timeout for unsolved instances, lowest score wins.)

Acknowledgment

Especial thanks to Foresta.IO https://www.foresta.io for the support on all these years of research.

References

[1] M. Heule, M. Jarvisalo, and M. Suda. Sat competition 2018. http://sat2018.forsyte.tuwien.ac.at

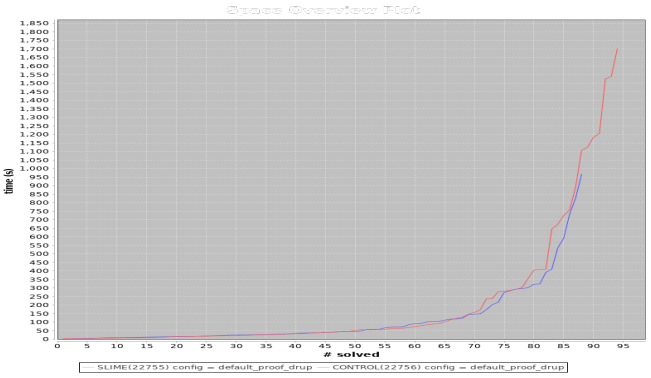


Fig. 1. SAT Race 2006 at 1800 seconds

TABLE I
SAT Race 2006

	SAT-VERIFIED	*UNKNOWN*	*UNSAT*
CONTROL	39 / 6200.9036	12 / 21600.5500	49 / 4073.3146
SLIME	40 / 13636.2266	6 / 10800.2000	54 / 5925.2390

PAR-2 SCORE: SLIME-41161.8656 vs CONTROL-53475.3182

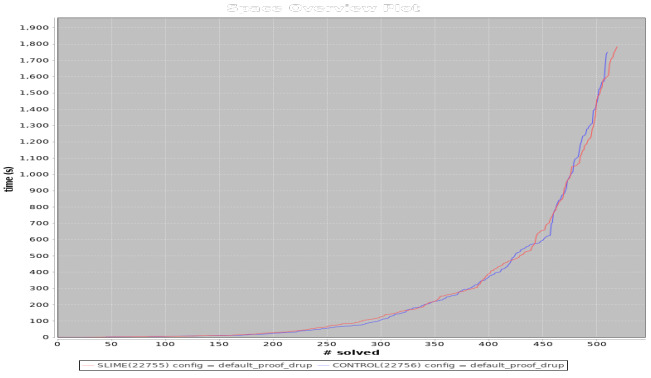


Fig. 2. SAT Race 2015 at 1800 seconds

TABLE II
SAT Race 2015

	SAT-VERIFIED	*UNKNOWN*	*UNSAT*
CONTROL	337 / 75335.6121	141 / 253807.9600	173 / 45136.8643
SLIME	316 / 72936.3637	132 / 237606.6200	203 / 63956.2390

PAR-2 SCORE: SLIME-612105.8427 vs CONTROL-628088.3964

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

SPARROWTOMERGESAT 2019

Adrian Balint
adrian.c.balint@gmail.com

Norbert Manthey
nmanthey@conp-solutions.com

***Abstract*—SPARROWTOMERGESAT is a combination of the solvers SPARROW and MERGESAT, and is heavily inspired by SPARROWTORISS. SPARROWTOMERGESAT is first trying to solve the problem with SPARROW+CP3, limiting its execution to $5 \cdot 10^8$ flips. If the formula could not be solved, the CDCL solver MERGESAT then tries to solve the problem.**

The SLS solver SPARROW is the same version as used in 2014 and 2018. The solver MERGESAT is the same solver as used plain in this years competition.

I. INTRODUCTION

While in 2014, where this solver combination was submitted for the first time, the benchmark was split in industrial and combinatorial families, recent competitions did not insist on this split any more. Last year, SPARROWTORISS performed amazingly well in the NOLIMIT track of the competition, when the number of solved formulas is considered. However, in the main track, SPARROWTORISS did not perform well. The major difference is due to the fact that SPARROWTORISS does not use SPARROW, as soon as a unsatisfiability proof is required, even though the input formula might be satisfiable. As there is no NOLIMIT track this year, the solver has been adapted.

The motivation of the combination of a SLS and CDCL solver can be found in [3], where formula simplification is used to boost the efficiency of SLS solvers on crafted families. The best found technique together with SPARROW represents the basis of our solver SPARROW+CP3. As SLS solvers cannot show unsatisfiability, we run a CDCL solver after a fixed amount of $5 \cdot 10^8$ flips, so that the overall solver behavior stays deterministic.

II. MAIN TECHNIQUES

SPARROW is a clause weighting SLS solvers that uses promising variables and probability distribution based selection heuristics. It is described in detail in [4]. Compared to the original version, the one submitted here is updating weights of unsatisfied clauses in every step where no promising variable can be found.

The used preprocessor CP3 is an extension of COPROCESSOR 2 [7], and received updates, but no changes since 2018.

The CDCL solver MERGESAT uses the MINISAT search engine [5], more specifically the extensions added in GLUCOSE 2.2 [1], [2], MAPLESAT [6] up until MAPLE_LCM_DIST_CHRONOBT [9].

Currently, no information is forwarded from the SLS solver to the CDCL solver. However, by using MERGESAT, a powerful CDCL engine is used, which is capable of producing unsatisfiability proofs. To motivate our approach, we now start with SPARROW+CP3, even if proofs are required. In case the SLS solver hits the step limit, we fall back to MERGESAT, which will produce the unsatisfiability proof.

III. MAIN PARAMETERS

SPARROW is using the same parameters as SPARROW 2011.

The configuration of CP3 has been tuned for SPARROW in [3] on the SAT Challenge 2012 satisfiable hard combinatorial benchmarks. The configuration used in 2019 is the same configuration used in the version of 2014 and 2018.

The details of MERGESAT are described in this years solver description [8].

IV. IMPLEMENTATION DETAILS

SPARROW is implemented in C. The solver MERGESAT is build on top of MINISAT 2.2, and many successfuly successors, and is implemented in C++.

V. AVAILABILITY

SPARROW is available at https://github.com/adrianopolus/Sparrow, and commit "satrace-2019" has been used.

The source of MERGESAT, as well as SPARROWTOMERGESAT is publicly available under the MIT license at https://github.com/conp-solutions/mergesat. The version with the git tag "satrace-2019" is used for the submission. The submitted starexec package can be reproduced by running "./scripts/make-starexec.sh -r satrace-2019 -s satrace-2019" on this commit.

REFERENCES

[1] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proc. 21st Int. Joint Conf. on Artifical Intelligence (IJCAI '09)*. Morgan Kaufmann, 2009, pp. 399–404.

[2] ——, "Refining restarts strategies for sat and unsat," in *Proceedings of the 18th international conference on Principles and Practice of Constraint Programming*, ser. CP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 118–126. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33558-7_11

[3] A. Balint and N. Manthey, "Boosting the Performance of SLS and CDCL Solvers by Preprocessor Tuning," in *Pragmatics of SAT*, 2013.

[4] A. Balint and A. Fröhlich, "Improving stochastic local search for sat with a new probability distribution," in *Proceedings of the 13th international conference on Theory and Applications of Satisfiability Testing*, ser. SAT'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 10–15. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14186-7_3

[5] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518.

[6] V. G. K. C. Jia Hui Liang, Chanseok Oh and P. Poupart, "MapleCOMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB," in *Proceedings of SAT Competition 2016*. [Online]. Available: http://hdl.handle.net/10138/164630

[7] N. Manthey, “Coprocessor 2.0: a flexible cnf simplifier,” in *Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing*, ser. SAT’12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 436–441. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31612-8_34

[8] ——, “Mergesat,” in *Proceedings of SAT Competition 2019*, 2019, submitted.

[9] V. Ryvchin and A. Nadel, “Maple_LCM_Dist_ChronoBT: Featuring Chronological Backtracking,” in *Proceedings of SAT Competition 2018*. [Online]. Available: http://hdl.handle.net/10138/237063

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

Topk-LC-Glucose

Jerry Lonlac
Research Center
IMT Lille Douai - University of Lille
Douai, France
jerry.lonlac@imt-lille-douai.fr

Englebert Mephu Nguifo
LIMOS, CNRS, UMR 6158
Clermont Auvergne University
Clermont-Ferrand, France
engelbert.mephu_nguifo@uca.fr

***Abstract*—This paper describes our CDCL SAT solver `Topk-LC-Glucose` which we submit to the SAT Race 2019.**

I. Introduction

Clause Learning [1], [2] is one of the most important components of a conflict driven clause learning (CDCL) SAT solver that is effective on industrial SAT instances. Since the number of learned clauses is proved to be exponential in the worst case, it is necessary to identify the most relevant clauses to maintain and delete the irrelevant ones. As reported in the literature, several learned clauses deletion strategies have been proposed. However the diversity in both the number of clauses to be removed at each step of reduction and the results obtained with each strategy increase the difficulty to determine which criterion is better. Thus, the problem to select which learned clauses are to be removed during the search step remains very challenging. Our SAT solvers `Topk-LC-Glucose` presented in this paper integrate a novel approach to identify the most relevant learned clauses without favoring or excluding any of the learned clause database cleaning strategies proposed, but by adopting the notion of dominance relationship among those measures. These solvers bypasse the problem of results diversity and reache a compromise between the measures assessments. Furthermore, they also avoid another non-trivial problem which is the number of deleted clauses at each reduction of the learned clause database.

II. Dominance relationship between learned clauses in Glucose

`Topk-LC-Glucose` was implemented on top of the solver `Glucose3.0` by integrating the learned clause database cleaning approach described in [3]. More precisely, this approach is obtained by selecting at each cleaning step of the learned clauses database, the *top-*k current undominated learned clauses (the k first Reference Learned Clauses [3]) according to a set of learned clauses relevant measures, and to delete all the learned clauses dominated by at least one of the *top-*k current undominated learned clauses. `Topk-LC-Glucose` solver avoids another non-trivial problem which is the amount of learned clauses to be deleted at each reduction step of the learned clauses database by dynamically determining the number of learned clauses to delete at each cleaning step. Dominance relationship between learned clauses is described in more detail in [3].

We submit to the SAT Race 2019 an implementation of our `Topk-LC-Glucose` solver integrating three learned clauses relevant measures in the dominance relationship: Size [4], [5]; that considers the shortest learned clauses as the most relevant, LBD [6]; that considers the clauses with the smallest LBD measure as the most relevant, and CSIDS [7]; that prefers the learned clauses most involved in recent conflict relationship.

III. Algorithm for Finding *top-*k learned clauses

During the search process, the CDCL SAT solvers learn a set of clauses which are stored in the learned clauses database $\Delta = \{c_1, c_2, ..., c_n\}$. At each cleaning step, we evaluate these clauses with respect to a set $\mathcal{M} = \{m_1, m_2, ..., m_k\}$ of relevant measures. We denote $m(c)$ the value of the measure m for the clause c, $c \in \Delta$, $m \in \mathcal{M}$. Since the evaluation of learned clauses varies from a measure to another one, using several measures could lead to different outputs (relevant clauses with respect to a measure). For example, consider the three learned clauses, c_1, c_2 and c_3 with their values on the three relevant measures *LBD*, *SIZE* and *CVSIDS* [7]:

- $SIZE(c_1) = 8$, $LBD(c_1) = 3$, $CVSIDS(c_1) = 1e^{10}$;
- $SIZE(c_2) = 6$, $LBD(c_2) = 5$, $CVSIDS(c_2) = 1e^{20}$;
- $SIZE(c_3) = 5$, $LBD(c_3) = 4$, $CVSIDS(c_3) = 1e^{30}$.

It comes from the previous example that c_1 is the best clause with respect to the LBD measure whereas it is not the case according to the evaluation of $SIZE$ measure which favors c_3. This difference of evaluations is confusing for any process of learned clauses selection. Hence, we can utilize the notion of dominance between learned clauses to address the selection of relevant ones.

Algorithm 1 starts by sorting the set Δ of learned clauses according to their degree of compromise [3]. It is easy to see that the first clause of Δ is not dominated, it is the *top*-1. So, at the beginning of the algorithm, we have at least one undominated clause. In step ind ($ind > 1$) of the outermost while-loop, the clause in position ind is compared to at most $ind - 1$ undominated clauses. As soon as it is dominated, it is removed, otherwise, it is kept as undominated clauses.

`Degree of compromise:` Given a learned clause c, the degree of compromise of c with respect to the set of learned clauses relevant measures $\mathcal{M}$ is defined by $DegComp(c) =$

Algorithm 1: `reduceDB_Dominance_Relationship`

Input: Δ: the learned clauses database; $\mathcal{M}$: a set of relevant measures; k: the number of reference learned clauses

Output: Δ t he new learned clauses database

```
sortLearntClauses() ;     /* by degree of compromise criterion */
ind = 1;
j = 1;
undoC = 1 ;        /* the number of current undominated clauses */
while ind < |Δ| do
    c = Δ[ind] ;          /* a learned clause */
    if c.size() > 2 and c.lbd() > 2 then
        cpt = 0 ;
        while cpt < undoC and ¬dominates(Δ[cpt], Δ[ind], M) do
            cpt++ ;
        if cpt >= undoC then
            saveClause() ;
            j++ ;
            undoC = min(k, j) ; /* minimum between j and k */
        else
            removeClause() ;
    else
        saveClause() ;
        j++ ;
        undoC = min(k, j) ; /* minimum between j and k */
    ind++;
return Δ ;

Function dominates(cMin: a clause, c: a clause, M)
i = 0;
while i < |M| do
    m = M[i] ;        /* a relevant measure */
    if m(c) ⪰ m(cMin) then
        return FALSE ;
    i++;
return TRUE ;
```

$\frac{\sum_{i=1}^{|M|}\widehat{m_i(c)}}{|M|}$, where $\widehat{m_i(c)}$ corresponds to the normalized value of the clause c on the measure m_i.

`dominance value`: Given a learned clause relevant measure m and two learned clauses c and c', we say that $m(c)$ dominates $m(c')$, denoted by $m(c) \succeq m(c')$, iff $m(c)$ is preferred to $m(c')$. If $m(c) \succeq m(c')$ and $m(c) \neq m(c')$ then we say that $m(c)$ strictly dominates m(c'), denoted $m(c) \succ m(c')$.

IV. Submitted versions

We submit two variants of `Topk-LC-Glucose` to the SAT Race 2019 and different scripts to start it with different parameters. Each variant with a different way for normalizing the set of learned clauses relevant measures which are defined in the follow.

How to normalize the values of the learned clauses?

For the two variants of our `Topk-LC-Glucose` solver submitted to the SAT Race 2019, we propose two ways for normalizing the values of the learned clauses. Given a learned clause relevant measure m and a learned clause c, we normalize the value of the clause c on the measure m using the two approaches described in the following.

Normalization of the learned clause values (approach 1):

- If m higher values are better, then $\widehat{m(c)} = \frac{1}{m(c)}$;
- If m smaller values are better, then $\widehat{m(c)} = \frac{m(c)}{nVars()}$, with $nVars()$ the number of variables of the Boolean formula.

Normalization of the learned clause values (approach 2):

- If m higher values are better, then $\widehat{m(c)} = \frac{m(c)}{M}$, where M is the upper bound of the learned clause values on the measure m;
- If m smaller values are better, then $\widehat{m(c)} = \frac{1}{m(c)}$.

For each variant of `Topk-LC-Glucose`, we submit two versions with respectively the parameter $k = 3$ (`Top3_Glucose`) and $k = 6$ (`Top6_Glucose`).

Acknowledgment

This research has been supported by Auvergne-Rhône-Alpes region and European Union through the European Regional Development Fund (ERDF) and University Clermont Auvergne. We also thank CRIL (Lens Computer Science Research Lab) Lab for providing them computing server and the authors of Glucose SAT solver for putting the source code available.

References

[1] J. P. M. Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Trans. Computers*, vol. 48, no. 5, pp. 506–521, 1999.

[2] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *38th Design Automation Conference (DAC)*, 2001, pp. 530–535.

[3] J. Lonlac and E. Mephu Nguifo, "Towards learned clauses database reduction strategies based on dominance relationship," *CoRR*, vol. abs/1705.10898, 2017.

[4] E. Goldberg and Y. Novikov, "Berkmin: A fast and robust sat-solver," *Discrete Applied Mathematics*, vol. 155, no. 12, pp. 1549 – 1561, 2007.

[5] S. Jabbour, J. Lonlac, L. Saïs, and Y. Salhi, "Revisiting the learned clauses database reduction strategies," *International Journal on Artificial Intelligence Tools*, vol. 27, no. 8, p. 1850033, 2018.

[6] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *IJCAI*, 2009, pp. 399–404.

[7] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, 2003, pp. 502–518.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

ZIB_Glucose: Luby Blocked Restarts and Dynamic Vivification

Marc Hartung
Zuse Institute Berlin
hartung@zib.de

***Abstract*—The solver ZIB_Glucose is based on Glucose 3.0 and mainly differs from its original in three implementation details. The dynamic restart heuristic was altered, a learned clause minimization (LCM) based on vivification was added, and the preprocessor was extended. These changes were made to boost the performance on satisfiable problems while maintaining performance on unsatisfiable ones.**

***Index Terms*—sat solver; restart heuristic; vivification; variable elimination**

I. Luby Blocked Restarts

Glucose uses a dynamic restart heuristic based on the LBDs of recently learned clauses [3]. The solver restarts when the last LBDs have a significantly higher average than the total LBD average. Additionally, when a solver reaches a significantly deeper search space, the LBD queue of the dynamic restart heuristic is flushed, which blocks a potential restart until the queue is full again, i.e., for a number n of conflicts equal to the queue's length. ZIB_Glucose delays restarts further by blocking potential restarts for $n \cdot luby(m)$ conflicts, where m is the number of blocked restarts and $luby()$ is the restart sequence (e.g. (1,1,2,1,1,2,4,1,1,...) for the power of 2) by the same name [2].

Additionally, we split the LBD queue in two parts where the first always contains the newest learned LBDs and the second contains the iteratively flushed LBDs of the first. A restart will then be triggered, when two conditions apply:

1) the average of both queue parts is larger than the total LBD average (equals the condition in Glucose 3.0) and
2) the average of the first queue part is larger than the total LBD average

The second condition allows the solver to stay in the search spaces, when recent LBDs are small and less recent LBDs are significantly larger. Glucose 3.0 allows per default an increase of the average LBD in the queue of 20 % before a restart is triggered. Due to increasing the number of conflicts per restart by introducing Luby in blocking and splitting the LBD queue, the increase of 20 % of LBD values is disabled in ZIB_Glucose.

The combination of both approaches leads to seldomly enforced long restart intervals and also rapid restarts, while trying to learn clauses of small LBD value. This behavior should increase the performance on satisfiable instances without significantly decreasing the performance on unsatisfiable problems.

This work received funding from the BMBF under grant 01IH15006C (HPSV).

II. Dynamic Vivification

In some test cases the LCM approach described by Luo et al. [7] using vivification [8] for learned clauses takes over 50 % of the total propagations. However, nearly no clause was minimized. To overcome this issue, we measure the impact of the minimization during solving and block, if necessary, the LCM. The impact is measured by tracking the number of failed vivifications f, the number of succeeded vivifications s and the rate r by which a clause was minimized on average ($r \in [0, 1]$). Then, the following condition should hold during the complete solving process:

$$\frac{s}{f} \cdot r + c \geq \frac{p_v}{p}$$

, where p is the total number of propagations, p_v the number of propagations used for the vivification and $c \in [0, 1]$ the minimal ratio spend for LCM. In the SAT Race we set $c = 0.01$,i.e., the solver spends at least 1 % on LCM propagations.

Since at some point further LCMs are blocked, the order in which the clauses are minimized is important. The missing key insight to properly solve this problem was given by Audemard et al. [4] by applying vivification first to the most active clauses.

III. Extended Preprocessing

The Glucose preprocessor is based on SatElite [5] and uses variable and clause elimination based on resolution and subsumption. Our version calls the elimination routine multiple times, where the first iteration is equal to the original preprocessor. The following iterations start with an empty subsumption queue and allow the variable elimination to remove a variable x, also when it increases the number of clauses with the restriction that the two clause sets (one containing all occurrences of x and the other all of $-x$) are constructed from the same set of variables. The intention behind this approach is to make already strongly connected parts of the SAT problem more efficient for unit propagation.

Additionally, the elimination is not disabled for huge problems (containing more than 480,000 clauses). Instead, we set a hard time limit of 150 seconds, which might be replaced by a deterministic counter in the future. This is also a quick fix for a performance bug, where the preprocessor exceeds the

computation time of 5000 s (e.g. happening for the 'barman-pfile08-032.sas.ex.15.cnf' of SAT competition 2016), which was to the best of our knowledge only fixed in the solver TopoSAT2 [1], [6].

ACKNOWLEDGMENT

We thank Niklas Eén and Niklas Sorensson for providing the MiniSAT implementation and also Gilles Audemard and Laurent Simon for the Glucose 3.0 implementation and their useful insights about SAT solving in their publications.

REFERENCES

[1] TopoSAT2 GitHub. https://github.com/the-kiel/TopoSAT2.

[2] Michael Luby Alistair, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.

[3] Gilles Audemard and Laurent Simon. Refining restarts strategies for SAT and UNSAT. In *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, pages 118–126, 2012.

[4] Gilles Audemard and Laurent Simon. Glucose and syrup: Nine years in the sat competitions. In *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, University of Helsinki, Department of Computer Science*, pages 24–25, 2018.

[5] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing*, pages 61–75, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[6] T. Ehlers, D. Nowotka, and P. Sieweck. Communication in massively-parallel sat solving. In *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*, pages 709–716, Nov 2014.

[7] Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 703–711, 2017.

[8] Cédric Piette, Youssef Hamadi, and Lakhdar Sais. Vivifying propositional clausal formulae. In *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*, pages 525–529, 2008.

BENCHMARK DESCRIPTIONS

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

Benchmark Selection of SAT Race 2019

Marijn J.H. Heule
Computer Science Department
Carnegie Mellon University, United States

Matti Järvisalo
HIIT, Department of Computer Science
University of Helsinki, Finland

Martin Suda
CTU/CIIRC
Czech Republic

The benchmarks for the SAT Race 2019 were drawn from a pool containing benchmarks that (i) were used in past competitive SAT events (SAT Competitions, SAT Races, and SAT Challenge 2012); and (ii) new benchmarks submitted to SAT Race 2019 (the descriptions for these benchmarks are provided in these proceedings). The benchmark suite consists of 400 benchmarks of which 200 are old and 200 are new.

As in recent SAT Competitions, the focus is on medium-to-hard benchmark formulas. As a consequence no old benchmarks that can be solved reasonably efficiently by a state-of-the-art solver of a decade ago were selected. The solver `MiniSAT` version 2.2 [1] was used as the representative solver from a decade ago. We ren MiniSAT on all benchmarks in the Main, Application and Crafted tracks of the prior competitive events and categorized the benchmarks into three buckets, easy, medium, and hard, based on the solver runtimes. A benchmark was categorized as "easy" if `MiniSAT` could solve it in 600 seconds, "hard" if `MiniSAT` could not solve it in 5000 seconds, and "medium" otherwise. For SAT Race 2019, we then selected 100 medium and 100 hard benchmarks from prior competitive events.

A weighted random selection was applied to each bucket. The weight for each benchmark depended on the track in which it was used in a prior competitive events. Benchmarks that were used in Industrial tracks were assigned weight 3, ones used in Crafted tracks weight 1, and ones used in Main tracks weight 2. These weights are motivated as follows: most benchmarks that were submitted to SAT Race 2019 were crafted. To make the combination of old and new benchmarks more balanced, we therefore prioritized industrial old benchmarks. Based on the weights, a random order of the benchmarks in each bucket was generated. From the medium bucket, the first 50 satisfiable and the first 50 unsatisfiable benchmarks wrt this order were selected. From the hard bucket, the first 100 benchmarks wrt this order were selected.

For SAT Competition 2018 it was mandatory to submit benchmarks in order to participate in the competition. This requirement was dropped for SAT Race 2019. Unfortunately, there were relatively few benchmarks submitted in 2019. As a consequence, it was challenging to include 200 new benchmarks, especially due to the restriction that at most 20 benchmarks could be selected from a SAT Race participant (in order to limit the influence of a participant on the results). By the submission deadline, the organizers received benchmarks from Joseph Bebel, Armin Biere, Jingchao Chen, Dieter von Holten, Norbert Manthey, Volodymyr Skladanivskyy, and Oleg Zaikin. (We kindly thank each of then for their contribution!) Most submissions contained between 10 and 20 instances. After the deadline we further asked Neng-Fa Zhou to generate new instances with his tool `Picat` [2] using instances of XCSP competition 2018. Additionally, the organizers used the benchmark challenges from a recent publication on matrix multiplication [3].

References

[1] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing*, E. Giunchiglia and A. Tacchella, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518.

[2] N.-F. Zhou, H. Kjellerstrand, and J. Fruhman, *Constraint Solving and Planning with Picat*, 1st ed. Springer Publishing Company, Incorporated, 2015.

[3] M. J. H. Heule, M. Kauers, and M. Seidl, "Local search for fast matrix multiplication," in *Theory and Applications of Satisfiability Testing – SAT 2019*, M. Janota and I. Lynce, Eds. Cham: Springer International Publishing, 2019, pp. 155–163.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

Harder SAT Instances from Factoring with Karatsuba and Espresso

Joseph Bebel
Department of Computer Science
University of Southern California
Los Angeles, California, United States of America
joseph.bebel@gmail.com

***Abstract*—These benchmarks were generated by the latest version of the ToughSAT software and are based on the hardness of factoring products of prime numbers. Instances based on products of primes are always satisfiable by exactly 2 assignments and, assuming widely accepted cryptographic assumptions, should be computationally hard to solve, with superpolynomial difficulty in the size of the primes used. The latest version of ToughSAT includes substantial improvements to the absolute and asymptotic complexity of the produced SAT instances, producing instances with significantly fewer variables and clauses for the same size prime numbers.**

***Index Terms*—factoring, SAT, karatsuba multiplication, fast integer multiplication, circuit optimization, boolean formula optimization**

I. Introduction

The ToughSAT project (https://toughsat.appspot.com/) by Joseph Bebel and Henry Yuen is designed to be an easy-to-use and easy-to-understand open source (https://github.com/joeintheory/ToughSAT) generator of computationally hard SAT instances. Specifically, it is a concrete implementation of several conceptually well-known strategies for hard SAT instance generation packaged as a web app and python script allowing quick and easy generation.

Integer Factoring is a well known problem in NP widely assumed to be hard in the average-case. This problem is not often necessary to compute in most practical applications and is most widely associated with cryptographic applications where instance sizes large enough to be computationally impractical to solve.

While ToughSAT is not the first software to generate hard SAT instances from Integer Factoring, a major goal of the project is to produce the "hardest possible" SAT instances from each Integer Factoring instance. Ideally, large instances of Integer Factoring (as measured by the number of bits in each prime factor) should be practically convertible to reasonably large SAT instances. For example, cryptographic size instances should produce instances which are not excessively large as to be unwieldy and impractical to generate, store, and load into SAT solvers (whether they are solvable by a SAT solver is a separate issue). In principle, assuming the the Exponential Time Hypothesis, solving SAT instances should require worst case running time exponential in the number of variables. Therefore, it is reasonable to suspect that "not many" additional variables are necessary to encode Integer Factoring into SAT instances.

The author gratefully acknowledges the support of the Annenberg Graduate fellowship and the ARCS Foundation Los Angeles Chapter.

ToughSAT has been improved towards this goal where smaller instances now produce more efficient boolean formulas using fewer variables and clauses in smaller DIMACS files. Interestingly, the reduction in variables anecdotally seems to produce instances which take longer to solve (compared to less efficient SAT instances generated from the same prime numbers).

II. Improvements to ToughSAT

The primary improvements to ToughSAT occurred in 2015 (the original ToughSAT was released in 2011) and have come from the use of the Karatsuba multiplication algorithm and the Espresso heuristic logic minimizer (https://ptolemy.berkeley.edu/projects/embedded/pubs/downloads/espresso/index.htm), along with some additional small optimizations.

The Karatsuba multiplication algorithm is a "fast integer multiplication" algorithm capable of multiplying n bit integers in time $O(n^{\log_2 3})$ time which is asymptotically better than the $O(n^2)$ time of classical integer multiplication. For our purposes, even though the numbers we are multiplying are rather small (20-30 bits per multiplicand, to obtain SAT instances that are solvable in practice) using the Karatsuba algorithm does produce smaller instances than the classical multiplication algorithm. While there are fast integer multiplication algorithms asymptotically superior to the Karatsuba algorithm, their large constant overhead likely would produce SAT instances larger than Karatsuba's in the 20-30 bit integer regime. It remains an open question whether asymptotically better integer multiplication algorithms would be superior in the cryptographic regime (greater than 384 bits per multiplicand), although such SAT instances are likely completely impractical as SAT solver benchmarks.

The instances are further shrunk substantially by use of the Espresso logic minimizer. The minimizer is not used directly in the construction of each SAT instance but rather was used to find minimal or at least smaller "building blocks" of each boolean functional unit used in Karatsuba's algorithm. For example, the initial implementation of Karatsuba's algorithm

may use basic units such as single bit adders, used in the form of CNF formulas on 5 variables that are satisfied on valid input/outputs of a 1 bit binary adder. This results in many intermediate variables, which for our purposes are wasteful. Instead, we can encode entire functional units (such as 8 bit adders and 6 bit multipliers) in CNF directly, with only the external input/output variables required (the internal intermediate variables being removed). Such CNF formulas of an 8 bit adder or 6 bit binary multiplier are rather large if written out in standard product of sums format (essentially describing the truth table of those functions) but can be optimized quite substantially with Espresso. The Espresso optimized functional units are used in our implementation of Karatsuba's algorithm.

In the small prime regime we are most interested in (at most 30 bit primes) this means we can avoid a substantial number of intermediate variables. For comparison, the 30 bit primes instance generated here needed only 988 intermediate variables to produce the 60 bit product, not far off from the approximately 900 intermediate variables required to simply even write down the partial products of the classical integer multiplication algorithm (completely ignoring the complex addition steps afterward).

III. Instances Generated

We generated a test instance constructed from two 23 bit prime numbers, using ToughSAT version 20190413, and used CryptoMiniSat 5.6.8 on default settings to estimate difficulty. CryptoMiniSat found a satisfying assignment in 313.2 seconds on a MacBook Pro (13-inch Retina Late 2013) with Intel Core i7-4558U 2800 MHz dual core CPU. Therefore, problems in this regime (22 to 26 bits) should be feasibly solvable using state-of-the-art SAT solvers given substantial CPU time.

28 distinct prime numbers (14 pairs of equal sized primes) were generated: 2 pairs of 22 bit primes; 4 pairs of primes of each of 23, 24, 25, and 26 bits; 1 pair of each of 27, 28, 29, and 30 bits. The primes were generated by OpenSSL 1.1.1b with the command "openssl prime -generate -bits b" where b is the number of bits required. Each pair was used to generate a single instance "20190413_toughsat_$b$bits_i.dimacs", where b is the number of bits in each prime factor and i is the index of the pair used. Each pair of primes uses distinct primes and no instances share prime factors.

In each DIMACS file, the target number (the product of the two primes) is given, along with the indexes of the variables encoding each prime factor. The input bits are encoded so that least-significant bit has smaller variable, so for example $p = x_{30}x_{29}\dots x_1$ and $q = x_{60}x_{59}\dots x_{31}$.

An observation: the 30 bit instance is nowhere near cryptographic size, however it may still be impractically large for some SAT solvers. The instance itself is quite reasonably sized (1108 variables, 27721 clauses, 897061 byte DIMACS file). In theory, subexponential time algorithms for Integer Factorization can solve products of 30 bit primes using almost no computational resources. However, to our knowledge there is no SAT solver in existence which implicitly or explicitly "encodes" the knowledge of number fields or other number theory to exploit any structure of the Integer Factoring problem. Therefore, it is not immediately obvious how exhaustive search can be avoided except for a constant number of bits. It is an interesting question to evaluate the extent to which a SAT solver can reliably find exploitable structure in factoring-sourced instances.

IV. Conclusion

The goal of the ToughSAT project is to use conceptually well known and understood methods to produce small, hard, SAT instances, as both an educational tool and a competitive hard SAT instance generator. Although by now, the optimizations and implementation details of ToughSAT are quite complex, the basic structure is still conceptually simple to describe using a widely understandable toolkit of concepts (integer factoring, fast integer multiplication) to produce competitively small and hard SAT instances. The reduction of intermediate variables has interesting effects on the perceived hardness of each instance. It would be interesting if a SAT solver could somehow reconstruct the functional units used in Karatsuba's algorithm to simplify its own analysis of each instance, or even implement higher order number theoretic methods.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

Arithmetic Verification Problems Submitted to the SAT Race 2019

Daniela Kaufmann Manuel Kauers Armin Biere
Johannes Kepler University Linz

David Cok
Safer Software Consulting

Multiplier Miters

In the benchmark description of our arithmetic challenge problems [1] submitted to the SAT Competition 2016, we have mentioned that there is another source of multiplier designs, which we could not retrieve back then. These circuits described in [2] were used in [3] and then synthesized and translated to AIGs in our related work [4]. Furthermore, the corresponding web-service "Arithmetic Module Generator" for generating the circuits (in Verilog) became recently available again at https://www.ecsis.riec.tohoku.ac.jp/topics/amg/. For the SAT Race 2019 we generated AIG miters and encoded them into CNF for interesting bit-widths 10, 12 and 14, where SAT solvers not using algebraic reasoning start to have a hard time. These benchmarks compare pairwise several multipliers with different architectures and characteristics. We also considered unsigned multipliers and a few signed multipliers (these are all $n \times n$ inputs to $2n$ bits outputs multipliers where signedness makes a difference). We compare two signed architectures "2cbpwtcl" and "2csparrc" with prefix "eq2..." which gives 6 signed benchmarks for bit-widths 10,12,14. The 12 unsigned multiplier architectures we compare are

bparcl, bparrc, bpctbk, bpdtlf, bpwtcl, bpwtrc,
sparcl, sparrc, spctbk, spdtlf, spwtcl, spwtrc

for bit-widths 10,12 and 14, which gives $396 = 3 \cdot 12 \cdot 11$ unsigned benchmarks (all with "eq..." but without "eq2", "btor" nor "ktsb" in their name).

Karatsuba Multiplication

As crafted benchmark we generated a bit-vector implementation of a single recursive step of the well-known Karatsuba multiplication algorithm. The implementation is then compared against a full multiplier of the same architecture (BTOR). We submitted only the three benchmarks "eqbtor10ktsb{10,12,14}*.cnf" for bit-widths 10,12 and 14.

The Crux of Multiplier Verification

During our work on multiplier verification we came across the issue that within a single column of a multiplier circuit (producing a certain output bit) the sum of the partial products can be permuted in an arbitrary order. Since adding up these partial products within a column needs adders of logarithmic size this summation requires bit-vector reasoning. In different multipliers these adders are ordered and grouped differently, which we conjecture to be the "crux" of multiplier verification on the bit-level.

To capture this problem we generated benchmarks which add up n bits with two input adder trees in a random order and grouping. The input bits are zero extended to m bits, which is the minimum number such that $2^m > n$. Then we generate two different random adder trees. Each tree consists of $n-1$ adders of bit-width m. The outputs of the two trees are compared, which is getting hard for standard SAT solvers on the CNF level at around $n = 30$ bits. We used 10 different seeds for $n = 20, \ldots, 32$ and thus submitted 130 benchmarks "cruxmiters{20,...,32}seed[0-9].cnf".

Integer Overflow Check

In program analysis of code similar to the following C program, the overflow check might yield hard bit-vector problems:

```
void *calloc (size_t a, size_t b) {
  if (((size_t)-1) / a < b) return NULL;
  return memset (malloc (a*b), 0, a*b);
}
```

Here is a corresponding SMT formula for this check

```
(set-logic QF_BV)
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert
  (not (=
    ((_ extract 63 32)
     (bvmul ((_ zero_extend 32) a)
    ((_ zero_extend 32) b)))
    (_ bv0 32))))
(assert
  (bvuge (bvudiv (bvnot (_ bv0 32)) a) b))
```

This is for a 32-bit machine. We generated 29 instances for bit-widths 20 to 48 called "davidcoqchallenge{20,...,48}.cnf". This problem is getting hard around 36 bits.

Supported by FWF, NFN Grant S11408-N23 (RiSE)

References

[1] A. Biere, "Collection of combinational arithmetic miters submitted to the SAT Competition 2016," in *SAT Competition 2016*, ser. Department of Computer Science Series of Publications B, T. Balyo, M. Heule, and M. Järvisalo, Eds., vol. B-2016-1. Univ. Helsinki, 2016, pp. 65–66.

[2] N. Homma, Y. Watanabe, T. Aoki, and T. Higuchi, "Formal design of arithmetic circuits based on arithmetic description language," *IEICE Transactions*, vol. 89-A, no. 12, pp. 3500–3509, 2006.

[3] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, "Formal verification of integer multipliers by combining Gröbner basis with logic reduction," in *DATE*. IEEE, 2016, pp. 1048–1053.

[4] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *FMCAD*, D. Stewart and G. Weissenbacher, Eds. IEEE, 2017, pp. 23–30.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

Generating SAT-based Cryptanalysis Instances for Keystream Generators

Oleg Zaikin
ISDCT SB RAS
Irkutsk, Russia
zaikin.icc@gmail.com

Stepan Kochemazov
ISDCT SB RAS
Irkutsk, Russia
veinamond@gmail.com

***Abstract*—This document briefly describes SAT encodings of cryptanalysis instances for seven keystream generators. Three of these generators are finalists of the eSTREAM project.**

I. Background

Keystream generator is a discrete function that given a short binary sequence (secret key) produces a binary sequence (keystream) of any required size. Keystream generators are used as primitives in stream ciphers. We consider a plaintext attack on keystream generators in the following formulation: based on the known fragment of keystream, it is required to find a secret key that was used to produce the fragment.

Cryptanalysis problems can be solved by SAT solvers [1]. We apply the SAT-based cryptanalysis to seven keystream generators. Four of them are described in [2]. In particular, two variants of the alternating step generator (ASG) [3], with 72-bit and 96-bit secret keys, are considered. They are further denoted as ASG-72 and ASG-96. Two modifications of ASG, MASG and MASG0 [4] with 72-bit secret keys are chosen. They are denoted as MASG-72 and MASG0-72. Also, Grain_v1, Mickey and Trivium are considered. Lengths of their secret keys are 160, 200 and 288 bits respectively. These three generators are finalists of the eSTREAM project [5]. This project was organized by European cryptological community and was aimed at identifying new fast and resistant stream ciphers. There were 7 finalists of this project in its second edition, we chose 3 of them.

II. Benchmarks

We submit 20 SAT instances made by the TRANSALG tool [6]. All secret keys are generated randomly. In Table I, the characteristics of the considered generators are presented.

In the benchmark set, there are 2 instances of ASG-72, 1 of MASG-72 and 1 of MASG0-72. All of them are satisfiable. Each of these 4 instances is simple enough to be solved by a state-of-the-art CDCL solver in about an hour. However, instances of ASG-96, Grain_v1, Mickey and Trivium are very hard, that is why weakened variants are considered for them in order to be able to solve them in reasonable time. First, for each of these 4 generators, 1 instance is constructed.

The research was partially supported by Council for Grants of the President of the Russian Federation (grant no. MK-4155.2018.9) and by Russian Foundation for Basic Research (grant no. 19-07-00746-a).

TABLE I
CHARACTERISTICS OF THE CONSIDERED KEYSTREAM GENERATORS.

Generator	Secret key (bits)	Keystream fragment (bits)
ASG-72	72	76
MASG-72	72	76
MASG0-72	72	76
ASG-96	96	112
Grain_v1	160	200
Mickey	200	250
Trivium	288	350

Then, 4 weakened instances are constructed based on it: 2 satisfiable and 2 unsatisfiable. For ASG-96, 2 weakened satisfiable/unsatisfiable instances are formed by assigning correct/incorrect values of the last 12 and 13 bits of the secret key. The incorrect values are chosen randomly. For other three generators, weakened instances were made in the same way – by assigning correct/incorrect values of the last bits of secret keys: for Grain_v1 – 104 and 105 bits; for Mickey – 146 and 147 bits; for Trivium – 142 and 143 bits. As a result, there are 8 unsatisfiable and 12 satisfiable instances in the benchmark set.

References

[1] F. Massacci and L. Marraro, "Logical cryptanalysis as a SAT problem," *J. Autom. Reasoning*, vol. 24, no. 1/2, pp. 165–203, 2000.

[2] O. Zaikin and S. Kochemazov, "An improved SAT-based guess-and-determine attack on the alternating step generator," in *ISC 2017*, ser. LNCS, vol. 10599, 2017, pp. 21–38.

[3] C. G. Günther, *Alternating Step Generators Controlled by De Bruijn Sequences*, ser. LNCS, 1988, vol. 304, pp. 5–14.

[4] R. Wicik and T. Rachwalik, "Modified alternating step generators," *IACR Cryptology ePrint Archive*, vol. 2013, p. 728, 2013.

[5] M. J. B. Robshaw and O. Billet, Eds., *New Stream Cipher Designs - The eSTREAM Finalists*, ser. Lecture Notes in Computer Science. Springer, 2008, vol. 4986.

[6] I. Otpuschennikov, A. Semenov, I. Gribanova, O. Zaikin, and S. Kochemazov, "Encoding cryptographic functions to SAT using TRANSALG system," in *ECAI 2016*, ser. FAIA, vol. 285, 2016, pp. 1594–1595.

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

Minimalistic Round-reduced SHA-1 Pre-image Attack

Volodymyr Skladanivskyy
Oslo, Norway
v.skladanivskyy@sophisticatedways.net

***Abstract*—We present a set of SAT instances that encode pre-image attack against SHA-1 reduced to 17 rounds. Our novel compact encoding minimises both number of auxiliary variables and number of clauses. We include the second set of instances mirroring the first one with several simplification pre-processing techniques applied to them. All our benchmarks are satisfiable with an expected single solution. We suggest that they represent a minimalistic version of the SHA-1 inversion SAT problem, hard enough to be challenging for modern day solvers.**

I. Introduction

Encoding of SHA-1 pre-image attack as a SAT problem is a well known method of obtaining SAT instances with desired level of hardness [1], [3], [4]. At the same time, selecting such instances can be challenging in practice as hardness increases rapidly with number of parameters and number of rounds going up [2]. Furthermore, even for low number of rounds the hardness varies by orders of magnitude depending on the choice of unfixed parameters and particular values of fixed parameters, i.e. bits of the hash and the message.

SHA-1 CNF encoding is often produced by translating its logic circuit representation using Tseitin transformation [1], [2], [5]. This approach when unoptimised introduces a large number of auxiliary variables and clauses. How much of this increase of size and complexity contributes to hardness, is unclear. An improvement of the same approach, our method [7] and tool [6] produces significantly more compact encodings than those published previously [5] thanks to algebraic optimisations and tailored imperative encoding for 32 bit modular addition.

We suggest that our benchmarks represent a minimalistic yet hard version of SHA-1 inversion SAT problem. For this competition, our aim is to benchmark performance of the various solvers on our compact encoding. In addition, it is to evaluate the impact of bespoke pre-processing on solvers' performance.

II. Benchmarks

Our benchmarks represent a single-block SHA-1. We encode the pre-image attack by fixing all of the hash bits and some of the message bits to pre-defined values. We keep unfixed a chosen number of the first message bits (message variables) in the sequence. We note that any practical attempt to find SHA-1 pre-image will involve constraining the message as well as the hash at least due to SHA-1 message padding requirement. Therefore, our variant of the SAT problem reflects complexity and hardness of a realistic pre-image attack.

The formulas are produced taking into account the fixed part of the message. That is, the corresponding message variables are excluded while encoding. We present two variants of each benchmark. The first variant has hash variables assigned using unit clauses and the second one is the result of applying a number of simplification techniques after the assignment, including unit propagation.

We chose the round-reduced version with 17 rounds because of practical considerations. In particular, instances with lower number of rounds are solvable in trivial time for any number of message variables while instances with higher number of rounds are larger and demonstrate higher variability of the solving time, making selection difficult.

Furthermore, we would like to note SAT instances with 160 message variables. With number of parameters equal to the number of constraints (160 hash bits), these instances are at the threshold of difficulty. Even with 17 rounds, solving them while practical, is beyond what is considered interesting for the purpose of this competition. That is, instances with 17 rounds are hard enough to be challenging for modern day solvers.

All our benchmarks are satisfiable by design. Satisfiability can be verified by solving after assigning the known values to the message variables. Also, all our benchmarks have less than 160 message variables. Therefore, they are expected to have single solution because of the uniformity of SHA-1.

We chose two solvers, glucose and cadical for verification and timing. We used the versions that participated in SAT Competition 2018, running on a workstation with performance characteristics comparable to StarExec cluster nodes, assuming that solvers use a single processor core.

III. Instances

Table I summarises the selected instances. All of them are solvable by the chosen solvers within 5000 seconds in our environment. We selected instances that are neither trivial nor very close to the timeout threshold. Also, we preferred instances with higher number of message variables. For each instance, we provide recorded solving time as a reference point.

With high variability of the solving time for even small changes of parameters, we were unable to identify any useful hypotheses with respect to the solving time other than glucose appeared to perform slightly better in the majority of our experiments.

TABLE I
SELECTED SAT INSTANCES

Instance Name	Message		Pre-proc.	CPU Time, s	
	ASCII	vars		glucose	cadical
sha1r17m63a	a	63	no	2882	519
sha1r17m63a_p	a	63	yes	415	837
sha1r17m66a	a	66	no	634	945
sha1r17m66a_p	a	66	yes	2691	129
sha1r17m67a	a	67	no	1174	289
sha1r17m67a_p	a	67	yes	1067	463
sha1r17m72a	a	72	no	1220	875
sha1r17m72a_p	a	72	yes	2768	218
sha1r17m75a	a	75	no	1281	116
sha1r17m75a_p	a	75	yes	1253	32
sha1r17m145ABCD	ABCD	145	no	1116	4099
sha1r17m145ABCD_p	ABCD	145	yes	352	2975
sha1r17m146ABCD	ABCD	146	no	2116	3940
sha1r17m146ABCD_p	ABCD	146	yes	409	1694
sha1r17m147ABCD	ABCD	147	no	1020	2673
sha1r17m147ABCD_p	ABCD	147	yes	1826	1311
sha1r17m148ABCD	ABCD	148	no	281	1956
sha1r17m148ABCD_p	ABCD	148	yes	218	2620
sha1r17m149ABCD	ABCD	149	no	2048	2274
sha1r17m149ABCD_p	ABCD	149	yes	1536	3824

REFERENCES

[1] D. Jovanović, P. Janičić, "Logical analysis of hash functions," *Frontiers of Combining Systems*, pp. 200–215, 2005. doi:10.1007/11559306_11

[2] V. Nossum, "SAT-based preimage attacks on SHA-1," Master's thesis, University of Oslo, 2012. [Online] Available: http://urn.nb.no/URN:NBN:no-33622

[3] V. Nossum, "Instance generator for encoding preimage, second-preimage, and collision attacks on SHA-1," *Proceedings of SAT Competition 2013*, pp. 119–120, 2013

[4] S. Nejati, J. H. Liang, V. Ganesh, C. H. Gebotys, and K. Czarnecki, "Adaptive Restart and CEGAR-based Solver for Inverting Cryptographic Hash Functions," *CoRR*, vol. abs/1608.04720, 2016. [Online]. Available: http://arxiv.org/abs/1608.04720

[5] Y. Motara, B. Irwin, "SHA-1, SAT-solving, and CNF," *Southern Africa Telecommunication Networks and Applications Conference (SATNAC)*, pp. 216–221, 2017

[6] V. Skladanivskyy, CGen, 2018. [Online] Available: https://github.com/vsklad/cgen. Accessed on: 02.04.2019.

[7] V. Skladanivskyy, "Tailored compact CNF encoding for SHA-1, " unpublished

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

Another SAT-Benchmark from Edge-Matching Puzzles

Dieter von Holten

***Abstract*—This document describes a set of benchmark-problems for SAT-solvers. The problems are yet another attempt to solve edge-matching puzzles like the notoriously difficult E2 puzzle. The underlying concepts and the structure of the model as well as the variants of the problem files are briefly explained.**

I. Introduction

A. 2-Phase Attack

Instead of trying to create *one* formula and solve the problem in *one* big step, we try a 2 phase-approach:

- Phase 1 tries to find 'Balanced Rotation Sets' (BRS). It rotates the tiles, until all edge-counts of all colors in north/south and east/west-direction are balanced, that they are equal. This is an invariant to all puzzle-solutions.
- Phase 2 then takes the fixed tiles-rotations and builds a new formula with a completely different structure, which tries to place the (non-rotating!) tiles on the grid, so that each edge has a peer of the same color.
  Phase 2 has been implemented: the solution of an E2-grade puzzle (size, number of colors, 5 hint-pieces) using a 'good' BRS as input can be found in a few minutes. Phase 2 is considered solved.

Unfortunately, the number of BRS for even mid-size puzzles is so large, that this approach is not practical.

B. local Balanced Rotation Sets

Other experiments brought the insight, that the 'color-counts must be balanced' invariant is true not just for the whole tile-set, but also for vectors. That is, in a row (or column), the number of east-edges (south-edges) of a color must be equal to the west-edges (north-edges) of that color. This constraint is much more precise. However, it is also much more expensive: instead of one large global tile-set we need to balance the color-counts in *nSize* rows and *nSize* columns. In addition, we now have to deal with the position of the tiles: we need to know, which tiles sit in row 5 or in column 12.
An incremental SAT-solver would produce these rotation-sets, which are then fed into Phase 2. It has been observed, that it takes a while to compute the first result, subsequent results are generated much quicker.

II. The Tile-model

Each tile has four bits to describe the rotation in '1-hot' notation. Furthermore, each tile has a 4-bit integer each to model the row and column-position. The position-range is limited from 0 to 15, in smaller puzzles the unused upper positions are prevented by blocking clauses. There are clauses, which further limit the valid positions (row, col) according to the tile-type:

- the 4 corner-tiles can have only positions $(0,0)$, $(0, nSize-1)$, $(nSize-1, 0)$ and $(nSize-1, nSize-1)$. The position enforces the rotation.
- the border-tiles can have only positions with forced rotation:
  in the northern border: $(0, 0 < col < nSize-1)$,
  in the western border: $(0 < row < nSize-1, 0)$,
  in the southern border $(nSize-1, 0 < col < nSize-1)$,
  and in the eastern border: $(0 < row < nSize-1, nSize-1)$
- the inner-tiles can have only positions $(0 < row < nSize-1, 0 < col < nSize-1)$.

Each tile has two decoder-circuits, which convert the binary form of the row- and column-position into '1-hot-format'; it is useful, to have the position-information available in both formats.

III. the Vector-model

A vector is a row or a column. It 'knows' it's index. The head (the west-end) and the tail (east-end) of a row are border-tiles, in between are $(nSize-2)$ inner tiles. In a row-vector we have two 'color-lines' for each color: one line holds the variables for the west-looking edges of this color, the other line holds the east-looking edges. A variable in a color-line is true, when the tile belongs to this vector (as indicated by tile-row-position and row-vector-index) and when the rotation indicates, that this edge shows west or east. The north and south edges are ignored in a row-vector. We have *BitCount*-constructs over the variables in the color-lines. The design of these *BitCount*-constructs is modelled after [Hackers Delight] and [Wikipedia]. The result is a bit-vector representing an integer. It is trimmed to three bits. Although we have up to 50 color-variables in a color-line, the actual size is limited to $nSize-2$, color-distribution of typical puzzles gives very small numbers of color-counts in a vector. The bit-count is used for these purposes:

- it is compared with it's peer bit-count of the same color in the same vector: they must be equal - this is *the* basic 'balanced color-count'-constraint.
- all bit-counts of one color over all vectors are added up. The total count must be equal to a known constant per color.
  this constraint is expensive and redundant, but it helps.

- all bit-counts over all colors in one vector are added up. The total is the number of relevant edges in a vector, it is compared to a known constant.
  this constraint is expensive and redundant, but it helps.

The clockwork described above is sufficient to provide 'locally balanced rotation sets'. However, when visually inspecting the results, some special cases show up:

- a tile with east and west edges of the same color sits in the row-vector. Obviously, it is balanced, but it is not useful in a solution.
- the inward looking edges of the head and tail have the same color. Obviously, this is balanced, but not useful in a solution.
- the inward looking edge of the head is green and the inward looking edge of the tail is red. There is one tile with a green west-edge and a red east-edge. Obviously, this is balanced, but not useful for a solution.

These cases are detected and prevented by additional constraints. This is sufficient to provide exact solutions for smaller puzzles. However, in larger puzzles, more complex invalidating patterns show up, which are difficult to avoid. The (not yet implemented) idea is to use an incremental solver to produce these results and check the quality of the solution outside of the SAT-solver.

IV. The Grid-model

The position-encoding of the tiles ensures, that each tile has only one position - but it does not prevent two tiles sitting on the same grid-cell. This is enforced by *AllDistinct*-constraints over all tile-positions. The two 4-bit integers of the row- and column-position are combined to an 8 bit grid-position and then these 8-bit grid-positions are pairwise given to bit-vector not-equal expressions [A.Biere technique]. The three tile-types (corner, border, inner) sit on disjoint areas on the grid, so we use 3 *AllDistinct*-constraints:

- one for the 4 corner-tiles
- one for the $4 \times (nSize - 2)$ border-tiles
- one for the $(nSize - 2)^2$ inner tiles

V. Symmetries

When you look at a 'locally balanced rotation set', you hardly can see any symmetry beyond the obvious rotation symmetry of the whole tile-set. But there is more: imagine a complete, valid puzzle solution. It is color-count-balanced in every aspect. Now cut out column-vector 3 and column-vector 11. Then swap the two column-vectors and put them back: 3 goes to 11, 11 goes to 3. Of course, this is no longer a valid puzzle-solution. But it is still completely balanced: no column was modified, and within each row, only two tiles have exchanged their position. No tile was rotated, all color-counts are the same as before. We can permute column-vectors with column-vectors, and rows with rows. We cannot swap the border rows and columns.
In a 16×16 puzzle, we have 14 inner vectors per direction; this gives us a total of $(14!)^2 = 7.6 \times 10^{21}$ permutations.
That means, the inner square of a puzzle-solution has billions of 'visual images' ! And each of them is just a few cpu-minutes away from the valid puzzle-solution.
The rotation-symmetry is removed by one or more hint-tiles. The vector-permuting symmetry is reduced by hint-tiles: the rows and columns, which contain one or more hint-tiles, cannot be swapped. Having $14-3 = 11$ free rows and columns leaves us with $(11!)^2 = 1.59 \times 10^{15}$ possible permutations.
It might be quite interesting to see how symmetry-aware tools would deal with this.

VI. The Challenge

We have these knobs to influence the difficulty of the problem:

- the size of the puzzle: larger puzzle, more variables, more bitcounters, adders, etc.
- the number of colors: the effect of phase-transition w.r.t size vs. colors is known, fewer colors for same puzzle-size means more edges per color, resulting in fewer, but longer bitcount-vectors.
- hint-pieces: we know the solution of the synthetic puzzles, we can place between 0 and n hint-pieces, more hint-pieces is easier.

VII. The Files

The zip-file contains these files:

- this description
- benchmark-files in DIMACS-format, same problem = same puzzle, but different parameters. The filename is build as:
  `b <year><month><problemId>-<size> x <size>`
  `c <colors> h <hintTiles><SAT|UNSAT>.cnf`

UNSAT is enforced by placing hint-tiles in conflicting positions.
Time shown is execution-time in wall-clock minutes with plingeling (version 17. May 2018) on a 4-core i5 3.1 GHz.

No	filename	problem	nSize	colors	hint-tiles	time
1	b1904P1-6x6c8h0SAT.cnf	P1	6	8	0	< 1
2	b1904P1-6x6c8h2SAT.cnf	P1	6	8	2	< 1
3	b1904P1-6x6c8h2UNSAT.cnf	P1	6	8	2	< 1
4	b1904P1-8x8c6h5SAT.cnf	P1	8	6	5	> 100
5	b1904P1-8x8c6h7SAT.cnf	P1	8	6	7	> 75
6	b1904P2-8x8c9h5SAT.cnf	P2	8	6	5	> 100
7	b1904P2-8x8c9h7UNSAT.cnf	P2	8	6	5	> 45
8	b1904P3-8x8c11h0SAT.cnf	P3	8	11	0	> 35
9	b1904P3-8x8c11h7SAT.cnf	P3	8	11	7	35
10	b1904P3-8x8c11h7UNSAT.cnf	P3	8	11	7	15

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

CBMC CNF Formulas 2019

Norbert Manthey
nmanthey@conp-solutions.com
Dresden, Germany

Michael Tautschnig
michael.tautschnig@qmul.ac.uk
Queen Mary University of London, United Kingdom

***Abstract*—The formulas that are generated by CBMC represent the NP problems that have to be solved by the tool to solve benchmarks from the Software verification competition. Currently, MINISAT 2.2 is used as a backend. With the submissions of this benchmark, we want to motivate research on SAT solvers in the direction of software verification.**

I. THE CBMC TOOL

CBMC [5] is a Bounded Model Checker for C and C++ programs. It supports C89, C99, most of C11 and most compiler extensions provided by gcc and Visual Studio. It also supports SystemC using Scoot. We have recently added experimental support for Java Bytecode [3].

CBMC verifies array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions. Furthermore, it can check C and C++ for consistency with other languages, such as Verilog. The verification is performed by unwinding the loops in the program and passing the resulting equation to a decision procedure.

CBMC comes with a built-in solver for bit-vector formulas that is based on MINISAT 2.2 [4]. Further, SAT solvers can be linked to CBMC via the IPASIR interface, and the solvers GLUCOSE 4.1 [1] and CADICAL [2] are supported as well. As an alternative, CBMC has featured support for external SMT solvers. Finally, the solver can also dump the CNF instead of running a solver. This feature has been used to generate the benchmark.

II. SAT COMPETITION 2019 BENCHMARK

The collected formulas have been dumped while running CBMC against the SV Comp 2019 benchmark. The timeout in this benchmark is 900 seconds. Within this time, a wrapper tries to solve a problem with multiple calls to the actual CBMC tool (see [5] for details). The CNFs that are considered for the benchmark are the CNF of the last CBMC call for a given input. Some of these CNFs are large, but can be solved easily, whereas others are small but more difficult to be solved.

From all categories of SV Comp where CBMC participated, we selected a subset of the categories, and measured the run time that CADICAL takes to solve them, and Any generated CNF formula is dropped as soon as CADICAL can solve it within 10 seconds.

III. AVAILABILITY

The source of CBMC can be found at https://github.com/diffblue/cbmc. For more details, please have a look at http://www.cprover.org/cbmc/. The used version of CBMC is: *cbmc-5.8-81-g9ae35de*, from https://github.com/diffblue/cbmc.git.

The used version of CADICAL is: *sc18-1-g58331fd*, from https://github.com/arminbiere/cadical.

REFERENCES

[1] G. Audemard and L. Simon, "Glucose and Syrup in the SAT'17," in *Proceedings of SAT Competition 2017*. [Online]. Available: http://hdl.handle.net/10138/224324

[2] A. Biere, "CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2018," in *Proceedings of SAT Competition 2018*. [Online]. Available: http://hdl.handle.net/10138/237063

[3] L. C. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtík, "JBMC: A bounded model checking tool for verifying java bytecode," in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, 2018, pp. 183–190. [Online]. Available: https://doi.org/10.1007/978-3-319-96145-3_10

[4] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518. [Online]. Available: https://doi.org/10.1007/978-3-540-24605-3_37

[5] D. Kroening and M. Tautschnig, "CBMC - C bounded model checker - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 389–391. [Online]. Available: https://doi.org/10.1007/978-3-642-54862-8_26

Proceedings of SAT Race 2019: Solver and Benchmark Descriptions, volume B-2019-1 of Department of Computer Science Series of Publications B, University of Helsinki 2019.

SAT Encodings for the Knights Hamiltonian Path Problem on Chessboards

Jingchao Chen
School of Informatics, Donghua University
2999 North Renmin Road, Songjiang District, Shanghai 201620, P. R. China
chen-jc@dhu.edu.cn

***Abstract*—The Knights problem is to ask whether a knight can visit all squares of an $n \times n$ chessboard on an admissible path exactly once. This problem is a special case of NP-complete graph problems. There exists a polynomial time algorithm to solve it. However, to the best of our knowledge, when $n > 20$, so far no SAT solver solve it if we use the distance encoding to translate it into a SAT problem.**

I. Introduction

The Knight's tour puzzle is defined as follows: Is it possible for a knight to start on some square and, by a series of admissible knight moves, visit each square on an $n \times n$ chessboard exactly once and return to the initial square of the tour? Admissible knight moves can be formalized by the edge set of the graph G.

The graph $G = (V, E)$ for the Knights tour puzzle on $n \times n$ chessboard is defined as the vertex set $V = \{(i, j) | 1 \leq i, j \leq n\}$ and the edge set E, describing the admissible moves of a knight, i.e.
$\{(i, j), (x, y)\} \in E \Leftrightarrow$
$|i - x| = 1$ and $|j - y| = 2$ or $|i - x| = 2$ and $|j - y| = 1$.

The Knight's tour problem is to ask whether there exists a cycle which contains each vertex exactly once. In general, such a cycle is called a Hamiltonian cycle. The Hamiltonian problem is NP-complete. However, the Knights tour problem can be solved in optimal sequential time $O(n^2)$ [1]. We translate it into a SAT problem by four encodings: CP encoding, bijection encoding, reachability encoding and distance encoding. The resulting SAT problem is not easy.

II. CP Encoding

The CP (Constraint Programming) encoding is used in Ref. [5]. It is based on the notion of MiniZinc's all_different [4]. The all_different constrains an array of variables to take different values. Its argument is an array of n integer variables, which has the form $[V_1, V_2, \ldots V_n]$, where $V_i (i = 1, \ldots, n)$ is an integer variable. Let E denote the set of edges in the Hamiltonian circle graph G. The CP encoding of the Hamiltonian circle on G may be described as follows.

$$\text{all\_different}([V_1, V_2, \ldots, V_n]) \quad (1)$$
$$V_i = j \Rightarrow (i, j) \in E \qquad 1 \leq i, j \leq n \quad (2)$$
$$\text{all\_different}([O_1, O_2, \ldots, O_n]) \quad (3)$$
$$O_1 = 1 \quad (4)$$
$$O_i = n \Rightarrow V_i = 1 \qquad 1 \leq i \leq n \quad (5)$$
$$V_i = j \Rightarrow Q_j = Q_i + 1 \qquad 1 \leq i, j \leq n \quad (6)$$

Constraints (1) and (2) ensure that each vertex has exactly one incoming edge. O_i denotes the order number of vertex i. The first visited vertex is 1, and the last visited vertex is n. This is done by constraints (4) and (5). Constraint (3) ensures that each vertex has a different order number. Constraint (6) means that if there is an edge from vertex i to vertex j, i.e., $V_i = j$, the order number of j is 1 more than that of vertex i, i.e., $O_j = O_i + 1$.

III. Bijection Encoding

The bijection encoding is based on Ref. [3]. The basic idea is to map the vertices to the positions in a Hamilton cycle, and assume that vertex 1 is in position 1. This encoding uses a two-dimensional 0/1 matrix M, whose entry M_{ij} is 1 iff vertex i is mapped to position j. Here is the CP bijection encoding for the Hamiltonian circle.

$$\sum_{j=1}^{n} M_{ij} = 1 \qquad 1 \leq i \leq n \quad (7)$$
$$\sum_{i=1}^{n} M_{ij} = 1 \qquad 1 \leq j \leq n \quad (8)$$
$$q = 1 + p\%n, (i, j) \notin E, M_{ip} \Rightarrow \neg M_{iq} \qquad 1 \leq i, j, p \leq n \quad (9)$$
$$q = 1 + p\%n, M_{ip}, M_{iq} \Rightarrow (i, j) \in E \qquad 1 \leq i, j, p \leq n \quad (10)$$

In the above encoding, $\%$ is a modular operator. q is the next position after p. This is computed by $q = 1 + p\%n$. Constraint (7) ensures that each vertex is mapped to exactly one position. Constraint (8) ensures that each position is mapped to exactly one vertex. Constraint (9) means that for each non-edge pair (i, j) if vertex i is mapped to position p, vertex j cannot be mapped to p's successor position. Constraint (10) means that if vertex i is mapped to position p and vertex j is mapped to p's successor, (i, j) is an edge in the Hamilton cycle.

IV. Reachability Encoding

The reachability encoding is used to convert answer set programs with loops into the SAT problem [2]. This encoding is to reduce the problem of finding a Hamilton cycle with n vertices to finding a Hamilton path from vertex 1 to vertex $n + 1$, where vertex $n + 1$ is a dummy vertex that has no outgoing edge to any vertex, and an incoming edge from each vertex i if $(i, 1)$ is an edge in G. This encoding uses a two-dimensional 0/1 matrix H of size $(n+1) \times (n+1)$. Each entry $H_{ij} \in H$ is 1 iff the edge (i, j) is included in the Hamilton path. In addition to matrix H, we use another 0/1 matrix R of size $(n + 1) \times (n + 1)$ to prevent sub-cycles. Each entry

$R_{ij} \in R$ is 1 iff there exists a Hamilton path from vertex i to vertex j. The reachability encoding consists of the following constraints.

$$\sum_{j=2}^{n} H_{1j} = 1 \tag{11}$$
$$\sum_{i=2}^{n} H_{i(n+1)} = 1 \tag{12}$$
$$\sum_{j=2}^{n+1} H_{ij} = 1 \quad 2 \le i \le n \tag{13}$$
$$\sum_{i=1}^{n} H_{ij} = 1 \quad 2 \le j \le n \tag{14}$$
$$H_{ij} \Rightarrow R_{ij} \quad 1 \le i, j \le n+1 \tag{15}$$
$$R_{ik} \wedge R_{kj} \Rightarrow R_{ij} \quad 1 \le i, j, k \le n+1 \tag{16}$$
$$\neg R_{ii} \quad 1 \le i \le n+1 \tag{17}$$

Constraint (11) ensures that vertex 1 has exactly one outgoing edge. Constraint (12) ensures that vertex $(n+1)$ has exactly one incoming edge. Except vertices 1 and $n+1$, each vertex has exactly one incoming edge and exactly one outgoing edge. This is done by constraints (13) and (14). Constraint (15) means that there is a path (R_{ij}) from vertex i to j if (i, j) is an edge in H. Constraint (16) describes the transitivity of the reachability relation.

V. Distance Encoding

The distance encoding uses an integer variable D_i to indicate how far vertex i is from vertex 1 in the Hamilton path. This encoding is similar to the reachability encoding. It uses a two-dimensional 0/1 matrix H of size $(n+1) \times (n+1)$, but replaces the two-dimensional matrix R with a one-dimensional matric D. The distance encoding consists of constrains (11)-(14) and the following constrains on the distance.

$$D_1 = 0 \tag{18}$$
$$D_{n+1} = n \tag{19}$$
$$H_{ij} \Rightarrow D_j = D_i + 1 \quad 1 \le i, j \le n+1 \tag{20}$$

Constraints (18) and (19) set the distance of vertices 1 and $n+1$ to 0 and n, respectively. Constraint (20) means that j's distance is 1 more than i's distance if (i, j) is an edge.

VI. Encoding of the Knights Hamiltonian Path Problem

We use the above four encodings to translate 20 Knights Hamiltonian path problem on $n \times n$ chessboard into SAT. 8 out of 20 Knights Hamiltonian problems are translated by CP encoding, which correspond to $n = 10$, 12, 14, 16, 18, 20, 22, 24, 26. 7 out of 20 Knights Hamiltonian problems are converted by distance encoding, which correspond to $n = 10$, 12, 14, 16, 18, 20, 22, 24. Two SAT problems are from reachability encoding, which correspond to $n = 20, 26$. One Knights Hamiltonian problem with $n = 16$ is translated by bijection encoding. In some cases, nn at-most-one constraint is encoded by the 2-product encoding with $O(n)$ clauses and $O(\sqrt{n})$ auxiliary variables [6]. The SAT problems produced by CP encoding and distance encoding seem to be hard. When $n > 20$, to our best knowledge, no SAT solver can solve them.

References

[1] Axe1 Conrad, Tanja Hindrichsb, Hussein MorsyC and Ingo WegenerdV: Solution of the knight's Hamiltonian path problem on chessboards, *Discrete Applied Mathematics*, pp. 125–134, 1994.

[2] Fangzhen Lin and Jicheng Zhao: On tight logic programs and yet another translation from normal logic programs to propositional logic. In IJCAI-03, pp. 853–858, 2003.

[3] Alexander Hertel, Philipp Hertel, and Alasdair Urquhart: Formalizing dangerous SAT encodings. In *Proceedings of SAT*, pp. 159–172, 2007.

[4] A MiniZinc Tutorial on Predicates and Functions, https://www.minizinc.org/doc-2.2.1/en/predicates.html

[5] NengFa Zhou and Håkan Kjellerstrand: Optimizing SAT encodings for arithmetic constraints. In CP, p.15, 2017.

[6] Jingchao Chen: A new SAT encoding of the at-most-one constraint. In Proc. of the 9th Int. Workshop of Constraint Modeling and Reformulation, 2010.

Solver Index

Benchmark Index

Author Index